Comparing Parallel Hardware Architectures for Visually Guided Robot Navigation

Wolfram Schenck\textsuperscript{1,2}\textsuperscript{*}, Michael Horst\textsuperscript{1}, Tim Tiedemann\textsuperscript{3}, Sergius Gaulik\textsuperscript{1} and Ralf Möller\textsuperscript{1}

\textsuperscript{1}Computer Engineering Group, Faculty of Technology, Bielefeld University, Universitätsstraße 25, 33615 Bielefeld, Germany
\textsuperscript{2}Simulation Lab Neuroscience — Bernstein Facility for Simulation and Database Technology, Institute for Advanced Simulation, Jülich Aachen Research Alliance, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
\textsuperscript{3}DFKI GmbH, Robotics Innovation Center, Robert-Hooke-Straße 1, 28359 Bremen, Germany

SUMMARY

Local visual homing methods are a family of algorithms for visually guided navigation on mobile robots. Within this family, the so-called Min–Warping algorithm yields very precise results but is rather compute-intensive. For this reason, we developed several implementations of this algorithm for different parallel hardware architectures (multi-core CPUs with SIMD extensions, GPUs, FPGA) to arrive at a fast and energy-efficient solution which is suited for real-time performance on mobile platforms with limited battery capacity. Because the Min–Warping algorithm is also well suited as a general benchmark, we carried out a comprehensive comparison study which includes both speed and real power measurements and covers both low-power processors and high-end devices. Our findings suggest that FPGAs offer the most energy-efficient platform for Min–Warping in the area of low-power processors, while GPUs take the lead in the area of high-end devices. However, as soon as the full capabilities of modern CPUs (like SIMD execution units and multiple hardware threads) are used, the speedup advantage of GPUs goes down to the single digit range. Copyright © 20XX John Wiley & Sons, Ltd.

Received …

KEY WORDS: GPU; FPGA; SIMD; performance analysis; power efficiency; robot navigation

1. INTRODUCTION

An important family of algorithms for visually guided robot navigation are local visual homing methods \cite{1,2}. Their purpose is to compute the movement direction ("home vector") for returning to a previously visited, nearby location solely based on interrelating the current view with a snapshot stored at the goal. The so-called Min–Warping method \cite{2} yields very precise home vectors,

\textsuperscript{*}Correspondence to: Email: w.schenck@fz-juelich.de

Contract/grant sponsor: The work was partially funded by the Helmholtz Association through the Helmholtz Portfolio Theme “Supercomputing and Modeling for the Human Brain”.

Copyright © 20XX John Wiley & Sons, Ltd.

\textit{Prepared using cpeauth.cls [Version: 2010/05/13 v3.00]}
but is computationally rather expensive because it works in a holistic manner on whole images instead of extracting and matching local features. However, mobile robots are usually equipped with comparatively slow low–power processors to extend battery life; this renders it difficult to arrive at the required update rates of the home vector for smooth movements in realistic scenarios.

For this reason we, examine in this paper three different implementations of the Min–Warping algorithm on different types of parallel hardware: multi–core–SIMD CPUs, graphics processing units (GPU), and field–programmable gate arrays (FPGA). Our goal is to identify in this way a fast and power–efficient approach for the implementation of Min–Warping and related algorithms. We furthermore think that the Min–Warping algorithm is well suited as general benchmark for parallel processor architectures because it is in most parts data–parallel, but with varying complexity of the computational structure and the memory access patterns. Therefore we include both low–power and high–end processors in our comparison study.

In general, comparisons between implementations for different hardware architectures are difficult and debatable. For this reason, we adopted the notion of the “competing programmer approach” from Alachiotis et al. [3]. For each implementation, different and especially experienced members of the author team were responsible for performance tuning (RM: SIMD on CPUs; SG/WS: GPU; MH/TT: FPGA) to ensure that each implementation actually measures up to the underlying potential of each compute device.

1.1. Modern CPUs: Multi–Core and SIMD

To fully exploit the potential of modern CPUs, one has to switch from the traditional single–threaded programming model and the use of scalar data types to a more flexible approach in which both multiple CPU cores and vectorized execution units are used wherever appropriate. Our “baseline implementation” of Min–Warping is based on the traditional approach, so that we can determine the speedup which is achieved by multi–threading (through OpenMP [4]) and vectorized data types (through SSE intrinsics [5] because we use x86 processors).

1.2. OpenCL for GPU–Computing

Today’s graphics processing units (GPU) are highly parallel processors which can also be used for general computations. They contain a large number of execution units for integer and floating-point operations. These units are organized within so–called multi–processors or SIMD–cores. The memory architecture is also hierarchical: The most important memory types are local and global memory. Each multi–processor has its own local memory store which is very fast and can be used as a programmer–managed cache. However, it is strongly limited in size. In contrast, global memory is large but slow. Usually, the main memory of the GPU (both local and global) is denoted as “device memory” and the main memory of the CPU as “host memory”. The large number of threads which run in parallel on a GPU puts strong pressure on the memory bandwidth. For this reason, an important programming challenge in GPU–computing is to carefully design memory access patterns so that accesses to global memory can be coalesced.

The most general approach for GPU programming is offered by OpenCL, a programming language and interface which runs on GPUs from different vendors [6]. For this reason, we used
OpenCL for a parallel implementation of the Min–Warping algorithm. OpenCL comprises a C–API for platform and runtime management on the host and a special programming language for the device code. In OpenCL nomenclature, a single unit of execution is called a “work–item” (corresponding to the otherwise often used term “thread”). A “work–group” contains a larger number of work–items which are scheduled together for execution on the same multi–processor.

1.3. Using FPGAs

CPU and GPU architectures are optimized to their specific needs: general computation or graphics processing, respectively. In contrast, field–programmable gate arrays (FPGAs) allow the design of circuits specifically for the target application: The single processing elements, their number, and the connections between them are laid out just as needed for the specific application and purpose. In this way, very power–efficient solutions can be achieved. However, since the design of the FPGA itself is not application–specific, the resulting FPGA–based implementations have some limitations. First, the size of the implementation is limited to the number of available components in the FPGA. Second, the maximum clock speed is usually much lower than the clock frequency of current CPUs. Third, the number of implemented processing elements in an FPGA design is less than the number of available processing elements in current GPUs.

1.4. Related Work

There exists a considerable amount of studies in which the performance of CPU and GPU implementations for specific tasks is compared (for example: [7, 8, 3, 9, 10, 11, 12, 13, 14, 15]). Even closer to our work are studies which include FPGAs or directly compare FPGAs to GPUs [16, 17, 18, 7, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]. The benchmarked applications are from many different fields like machine learning [9, 10, 15], neural modeling [11], optimization [21, 13], numerical algorithms [7, 26], image and video processing [16, 22, 24, 25, 26, 27, 30], computer tomography [20, 31], molecular sequencing [3], financial simulations [19], encryption and decoding [23, 32], or analog circuit simulation [17].

The contest between CPU and GPU implementations is normally won by the GPU side. However, the full capabilities of modern CPUs are often not exploited which leads to unfair comparisons. Exceptions are studies in which at least multi–threading is applied to occupy all CPU cores [17, 8, 24, 12, 14], or in which both multi–threading and vectorized code are used [16, 7, 20, 23, 3, 32, 15]. In such a setting, Alachiotis et al. [3] and You et al. [15] no longer observe a consistent advantage for the GPU implementation.

The comparison between GPUs and FPGAs does not result in a clear picture. In half of the studies we consider here the FPGA outperforms the GPU — at least for some of the reported problems and problem sizes [16, 17, 19, 21, 23, 24, 27, 30, 31]. This is also true if we only examine applications which require real–time performance (GPU as winner: [22, 25, 26]; FPGA as winner: [27, 30]). In the only comparison addressing truly embedded devices [27] the FPGA has the lead (GPU Nvidia ION 2 vs. FPGA Xilinx Spartan–3).

A rather small number of these studies also includes considerations regarding energy consumption based on power specifications supplied by the manufacturers [21, 24, 7, 27]; even more rare are
studies with actual power measurements [22, 8, 19, 31, 32]. The general consensus is that FPGAs require the smallest amount of energy given a specific computational task.

In our study on Min–Warping we aim on a rather unexplored area by (1) including an advanced CPU implementation with multi–threading and vectorization in the performance comparison and (2) carrying out real power measurements for the assessment of the energy consumption. Furthermore, our study addresses both low–power compute devices (incl. an FPGA) and high–end devices. We are very well aware that comparison studies like this are easily biased by several factors at the programming and hardware level. Ideally, one would normalize factors like development time, programming skill, quality of drivers and run–time libraries, power consumption, manufacturing process, release year of the hardware, etc. We achieve this at least partly by the above–mentioned “competing programmer approach”, by carrying out real power measurements, and by using CPUs and GPUs which were released within the years 2010/11. In comparison, the FPGA within our study (Altera Stratix-II) is rather outdated. However, this is taken into account for the interpretation of the obtained results.

The remainder of the paper is organized as follows: First, the Min–Warping algorithm is described (Sect. 2), afterwards the specific implementations (Sect. 3). In Sect. 4, the results regarding performance and power efficiency are presented, and finally discussed in Sect. 5.

2. THE MIN–WARPING METHOD

Local visual homing methods take two images recorded at different locations in space and determine the relative movement parameters between the camera postures at these locations. When visual homing is used to reach a previously visited place, the image taken at the current position $C$ is called current view (CV), the image at the goal position $S$ snapshot (SS). Typically, these images are low–pass filtered panoramic grayscale images which depict a 360 degree view from the robot’s perspective (see Fig. 1).

The Min–Warping method tries to find the relative movement direction $\alpha$ and change in orientation (compass) $\psi$ from the snapshot to the current view (see Fig. 2, left). Based on these two parameters, the home vector pointing from the current position to the desired goal position can be derived. It is a holistic method which considers the entire image and treats each image column as a single landmark instead of extracting separate features. The surrounding environment is assumed to be static, and movements are restricted to a 2D plane.

The basic idea behind warping methods is to distort one of the images according to a given combination of movement parameters and compare it with the other image. This requires an exhaustive search over $\alpha$ and $\psi$. The parameter combination $(\hat{\alpha}, \hat{\psi})$ yielding the best match is assumed to correspond to the correct movement.
Figure 2. Derivation of the warping equations (left: top view, right: side view). A robot moves from a snapshot location $S$ to a current location $C$, orientated as indicated by the thick arrows. The movement is described by the parameters $\alpha$, $\psi$, and $d$ (the latter of which is only implicitly addressed in Min–Warping). The azimuth $\Theta$ of a landmark feature at $L$, located at height $h$ above the camera plane, changes to $\Theta' = \Theta + \delta$, its elevation changes from $\gamma$ to $\gamma'$. Figure taken from [33] (with permission from Elsevier), left part (A) after [1].

For Min–Warping, the distortion can be described as follows: First, we assume that the two images are aligned along the presumed movement direction (in this case, $\alpha$ and $\psi$ are both zero). A landmark $L$ can be seen from $S$ at an angle $x$. When moving towards the current view $C$, this landmark column is shifted by $y$ toward the image’s focus of contraction (FOC). The amount $y$ that the column is shifted does not only depend on $x$, but also on the ratio of the covered distance and the distance to the landmark — which is usually unknown when using only intensity images. In addition, $y$ is constrained by the following limitations (the following explanation is for landmarks to the left of the movement direction; respective conditions hold for the other side):

- $y$ is greater or equal zero. This results from the fact that the column is shifted towards the FOC.
- $x + y$ is smaller than $\pi$. This means that the column cannot be shifted past the FOC. $x + y = \pi$ implies $x = \pi$ and $y = 0$.
- For a landmark in movement direction ($x = 0$), $y$ is ambiguous. It can switch from 0 to $\pi$ when moving past the landmark.

In the general case in which $\alpha$ and $\psi$ hold arbitrary values, the landmark appears at an angle $\Theta$ before the movement at $S$, and at an angle $\Theta' = \Theta + \delta$ afterwards at $C$ (Fig. 2, left). To align the images relative to the movement direction, they are rotated by $\alpha$ (SS) or $\alpha - \psi$ (CV), resulting in $x = \Theta - \alpha$ and $y = \Theta' - \Theta + \psi = \delta + \psi$.

During the search for the best parameter combination ($\hat{\alpha}$, $\hat{\psi}$), the $\alpha$ and $\psi$ values are varied systematically. Given specific values ($\alpha^*$, $\psi^*$), the goal is to identify for every landmark column in the snapshot SS the most similar landmark column in the current view CV, because similarity signals that both columns depict the same landmark. For a specific column in SS (corresponding to the angle $x$, or respectively $\Theta$), all possible landmark shifts $y$ in CV are examined (subject to the constraints...
Figure 3. Left: Computation of a single scale plane (quadratic image in the middle) from a snapshot (SS) and a current view (CV). Note that snapshot and current view are cyclic in horizontal direction and scale–plane images are cyclic in both horizontal and vertical direction. Dark pixels in the scale plane indicate a small distance between the respective image columns in the SS and CV, light pixels a large distance. Right: The same scale plane as on the left side, but this time \( j_\delta \) is varied in vertical direction instead of \( j_\Theta \) (\( j_\Theta = (j_\Theta + j_\delta) \mod W \)) (Figure adapted from [2]; with permission of Springer). While the left form of the scale plane provides a more intuitive understanding, the right form corresponds to the scale template as shown in Fig. 5. Both forms are used at the programming level, depending on the implementation.

Figure 4. Stack of scale planes with five different pre-set scaling factors \( \sigma_k \). Figure adapted from [2] (with permission of Springer).

mentioned above, and discretized to horizontal pixel indices in CV), and the Euclidean distance between the corresponding image columns in SS and CV is computed. The minimum of all distance values over \( y \) is determined, and these minimum distances are accumulated over all image columns in SS, resulting in a total distance value which is specific for the given parameter combination \((\alpha^*, \psi^*)\). Finally, as result of the iteration over \( \alpha \) and \( \psi \), the parameters \( \hat{\alpha} \) and \( \hat{\psi} \) corresponding to the minimal total distance (which indicates the best overall match) are the result of the Min–Warping method. The estimated direction \( \hat{\beta} \) of the home vector pointing from C to S can finally be found as \( \hat{\beta} = \hat{\alpha} - \hat{\psi} + \pi \).
To enhance the precision of the algorithm, the vertical scaling resulting from moving towards or away from an object (see Fig. 2, right) is considered when computing the distance between image columns. The scale factor $\sigma$ can be approximated using $\sigma = \gamma = \frac{\tan \gamma}{\tan \gamma} \approx \frac{\gamma}{\gamma}$. Pixels inside a column are shifted (relative to the image horizon) simply by scaling with $\sigma$. Using the law of sines in Fig. 2 (left), $\sigma$ can also be expressed using $x$ and $y$:

$$\sigma = \frac{\sin x}{\sin (x + y)}$$ (1)

For an efficient implementation, the distance values between each pair of columns can be precomputed for a discrete set of scale factors $\sigma_k$, yielding for each scale factor a so-called scale plane (quadratic images in Fig. 3). To generate a single scale plane for panoramic images with width $W$, $W \cdot W$ Euclidean distances between image columns have to be computed, iterating over $\Theta$ and $\delta$ (or $\Theta'$). A full scale–plane stack $D$ is shown in Fig. 4 for $K = 5$ different scale factors.

After the precomputation of the scale–plane stack, the main part of the Min–Warping algorithm is the search phase (as described above), in which four nested loops are used to iterate over the warping parameters $\alpha$ and $\psi$ and over the landmark position $x$ and shift $y$. In our notation (see Table 1), the discretized angles (with pixel accuracy) are denoted as $j_x$, $j_y$, $j_\alpha$, and $j_\psi$ (these discretized angles run from $0 \ldots W - 1$ corresponding to a full 360 degree range). Because every combination of $j_x$ and $j_y$ enforces a specific scale factor $\sigma$ (according to (1)), it is possible to precompute a scale template which defines for every possible combination of $j_x$ and $j_y$ the index of the corresponding scale plane which has to be used to retrieve the precomputed distance value. The entries of the scale

---

Figure 5. Scale template for panoramic images with a width of 24 pixels (just for illustration; the actually used width amounts to at least 288 pixels in our studies). On the left side, the $\sigma$ values according to (1) are shown for each combination of $j_x$ and $j_y$, and on the right side the corresponding scale–plane indices for a stack with $K = 8$ scale planes (the template on the right is denoted as “planetmpl$[j_y][j_x]$” in line 17 of Fig. 6). Note the empty columns in which no valid values exist (directly into movement direction and exactly opposite to the movement direction). Figure taken from [2] (with permission of Springer).
Table I. Important parameters of the Min–Warping algorithm.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W, H$</td>
<td>Width and height of panoramic images</td>
</tr>
<tr>
<td>$\alpha, \psi$</td>
<td>Warp parameters</td>
</tr>
<tr>
<td>$\Theta, \Theta'$</td>
<td>Azimuth of landmark seen from snapshot / current location</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Change in azimuth of a landmark ($\delta = \Theta' - \Theta$)</td>
</tr>
<tr>
<td>$x$</td>
<td>Azimuth of landmark relative to presumed movement direction ($x = \Theta - \alpha$)</td>
</tr>
<tr>
<td>$y$</td>
<td>Change of azimuth of landmark relative to presumed movement direction ($y = \delta + \psi$)</td>
</tr>
<tr>
<td>$j_\alpha, j_\psi, j_\Theta, j_\Theta', j_x, j_y$</td>
<td>Discrete pixel indices corresponding to $\alpha, \psi, \Theta, \Theta', \delta, x, y$ (running from $0 \ldots W - 1$ corresponding to a full 360 degree range)</td>
</tr>
<tr>
<td>$n_\alpha, n_\psi$</td>
<td>Number of steps for iterating over warp parameters $j_\alpha, j_\psi$</td>
</tr>
<tr>
<td>$s_\alpha, s_\psi$</td>
<td>Corresponding step size ($s_\alpha = W/n_\alpha; s_\psi = W/n_\psi$)</td>
</tr>
<tr>
<td>$i_\alpha, i_\psi$</td>
<td>Parameter indices corresponding to $j_\alpha, j_\psi$ values ($j_\alpha = i_\alpha \cdot s_\alpha; j_\psi = i_\psi \cdot s_\psi$)</td>
</tr>
<tr>
<td>$\gamma, \gamma'$</td>
<td>Vertical angles of landmark features above the horizon seen from snapshot / current location</td>
</tr>
<tr>
<td>$\tilde{j}<em>\alpha, j</em>\gamma, \tilde{j}<em>\psi, \tilde{j}</em>\gamma'$</td>
<td>Discrete pixel indices corresponding to $\gamma, \gamma'$</td>
</tr>
<tr>
<td>$SS[j_\Theta][j_\alpha], CV[j_\gamma][j_\Theta]$</td>
<td>Snapshot and current view</td>
</tr>
<tr>
<td>$K$</td>
<td>Number of scale planes</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>Scale factor</td>
</tr>
<tr>
<td>$\sigma_k, \tau_k$</td>
<td>Scale factor of scale plane $k$, scale–factor threshold</td>
</tr>
<tr>
<td>$D[k][j_\Theta][j_\alpha]$</td>
<td>Scale–plane stack ($k \in 0 \ldots K - 1$ is the scale–plane index)</td>
</tr>
</tbody>
</table>

Template are determined by applying thresholds $\tau_k$ to match the scale factors $\sigma$ to the correct scale–plane index $k$ (see Fig. 5). The computation time to generate the scale template is negligible and will not be considered in the performance study.

Because $y$ is constrained depending on $x$ as explained above, the loop over $j_\psi$ can be restricted to valid entries in the corresponding column in the scale template. Regarding the outer loops, the iteration over $j_\alpha$ and $j_\psi$ is executed with a pre–defined number of steps $n_\alpha$ and $n_\psi$, respectively, chosen such that the image width $W$ is an integer multiple of both parameters. Pseudo code for the complete Min–Warping algorithm is presented in Fig. 6, in which the loop indices for iterating over $j_\alpha$ and $j_\psi$ are denoted as $i_\alpha$ and $i_\psi$ (see also Table I).

3. IMPLEMENTATIONS

3.1. Serial Implementation

The serial CPU implementation works in a straightforward way along the pseudocode description in Fig. 6, including as optimization early termination of the $j_x$–loop (lines 22-24 in Fig. 6). Images and scale planes are stored in integer format (one 32–bit integer value per pixel). For the scale template (see Fig. 5) a compressed array is used, leaving out any empty entries.
1: compute scale-plane stack $D$
2: $\hat{d} = \text{large}$
3: for $i_\psi = 0 \ldots n_\psi - 1$
4: $j_\psi = \text{index}\{\psi[i_\psi]\}$
5: for $i_\alpha = 0 \ldots n_\alpha - 1$
6: $j_\alpha = \text{index}\{\alpha[i_\alpha]\}$
7: $d = 0$
8: // index $j_x$ corresponds to $x$
9: for $j_x = 0 \ldots W - 1$
10: // index $j_\Theta$ corresponds to $\Theta = x + \alpha$
11: $j_\Theta = (j_x + j_\alpha) \mod W$
12: if template column $j_x$ is not empty
13: $m = \text{large}$
14: foreach $j_y$ in template column $j_x$
15: // index $j_\delta$ corresponds to $\delta = y - \psi$
16: $j_\delta = (j_y - j_\psi) \mod W$
17: $k = \text{planetmpl}[j_\delta][j_x]$
18: $v = D[k][j_\delta][j_\Theta]$
19: $m = \text{min}\{m, v\}$
20: endforeach
21: $d = d + m$
22: if $d > \hat{d}$
23: break
24: endif
25: endif
26: endfor
27: if $d < \hat{d}$
28: $\hat{d} = d$
29: $\hat{(i_\psi, i_\alpha)} = (i_\psi, i_\alpha)$
30: endif
31: endfor
32: endfor
33: determine $\hat{\psi}, \hat{\alpha}$ from $\hat{(i_\psi, i_\alpha)}$

Figure 6. The Min–Warping algorithm. Note that the loop in line 14 only includes non–empty template elements in Fig. 5 (right). Presentation adapted from [2] (with permission of Springer).

3.2. Parallel Implementation for CPUs

The parallel implementation of Min–Warping for CPUs relies on two principles of parallelization: vectorization and thread parallelism. Vectorization is achieved by using SSE2 (streaming SIMD extensions) instructions available in many processors of the Intel family, plus instructions from SSSE3 (supplemental streaming SIMD extensions). We decided to use “SSE intrinsics” [34] in the C/C++ language rather than assembler code. The resulting code is not only better readable than assembler, but also offers the compiler the chance to perform code optimization. The parallelization was done “by hand”; so far we haven’t tested automatic code parallelization [35]. Thread parallelism relies in our implementation on the OpenMP framework.

3.2.1. Computation of the scale–plane stack As outlined in the previous section, the scale–plane stack is computed in the first phase of the Min–Warping algorithm. Each entry in a single scale plane
is computed as Euclidean distance \( E \) between the corresponding columns in the snapshot (vector \( a \)) and current view (vector \( b \)) (see Fig. 3):

\[
E(a, b) = \|a - b\| = \sqrt{\|a\|^2 + \|b\|^2 - 2a^Tb}
\]

In the parallel CPU implementation, the rightmost form of the Euclidean distance is applied, including precomputation of \( \|a\|^2 \) and \( \|b\|^2 \). Obviously, the scale–plane computation can be parallelized in a trivial way since the computation of an entry in the scale plane is independent of all others. Moreover, the structure of the problem (dense arrays) is very regular and fits well to a SIMD implementation.

Essentially, three parts of the computation can be parallelized:

1. the computation of the squared vector lengths (\( \|a\|^2, \|b\|^2 \)),
2. the computation of the scalar products (\( a^Tb \)) (this step dominates computational effort), and
3. the addition of the terms under the square root.

The computation of squared vector lengths and scalar products requires multiplications and additions. We decided for an integer implementation because SSE2/SSSE3 only allows for four–fold parallelism in the float domain, while the corresponding integer operations can operate on eight 16–bit values in parallel; this bit depth is sufficient for Min–Warping. The SSE2 extension offers multiplication of 16–bit integer values in two 128–bit data words with the assembler instruction pmulhuw. Addition would also be available for 8–bit operands, but we decided to use a 16–bit representation throughout to avoid format conversions.

For the computation of the squared vector lengths, eight squared vector lengths are computed in parallel and the result is stored in memory before starting the next computation. For the computation of the scalar products we tested different versions. In the fastest one, the loop order is such that temporary results for eight scalar products are written to memory, before the same computation is performed for another eight scalar products. For this reason, the width \( W \) of the images has to be an integer multiple of eight. Moreover, the memory allocated (for images, scale planes, and intermediate data) has to be aligned to 16 bytes for efficient SSE2 access.

In the final part of the scale–plane computation, the overall distance measure is computed from vector lengths and scalar products; SSE2 instructions are used as well for this purpose. Note that a 64K word lookup table is used to compute all square roots. In this part we also rearrange the scale–plane data from the \( \Theta' \) vs. \( \Theta \) form (which is better suited for the SIMD computation of the scale–plane stack) into the \( \delta \) vs. \( \Theta \) form (as shown in Fig. 3 on the right) required for the second phase of Min–Warping. Furthermore, the scale–plane rows are shuffled as described in the next section.

In addition to parallelization through vectorization, the OpenMP framework is used to achieve parallelism at the thread level for the computation of the scalar products. For this purpose, the outer loop over the columns of the snapshot image is split up between as many threads as hardware threads are available through simultaneous multithreading.

3.2.2. Search phase While the first phase of Min–Warping could be parallelized in a trivial way, the second phase requires a rearrangement of the scale–plane stack so that the data is accessible in consecutive 8–word chunks (required for SSE2 load instructions). For this purpose,
our implementation shares the idea suggested in [36] to compute the match values in parallel for a set of (here eight) consecutive values of the parameter \( j_\alpha \). Thereby, the match value for eight positions of the scale template on the scale–plane stack can be computed simultaneously.

The major obstacle to overcome in the SSE2 implementation was the fact that the search space (over \( j_\alpha \) and \( j_\psi \)) is not discretized in steps which correspond to the one–pixel distance between image columns, but with a coarser resolution of typically 2 to 4 pixels. This coarser resolution reduces the effort for the search (in quadratic dependence on the step width). If, for example, the image width \( W \) is 288, and the step width for \( j_\alpha \) is \( s_\alpha = 3 \), we only search through \( n_\alpha = W/s_\alpha = 96 \) values for \( j_\alpha \) (from \([0, W)\)). If we consider the very first parallel computation step of the algorithm for the values \( j_\alpha = \alpha_0 \ldots \alpha_7 \), column 0 of the scale template positioned at \( \alpha_0 \) considers column 0 of the scale–plane stack, column 0 of the scale template at \( \alpha_1 \) considers column 3 of the scale–plane stack, column 0 of the scale template at \( \alpha_2 \) considers column 6 etc.; see Fig. 7, top. We can process all these elements in the same way in parallel (since the template entry is the same for all of them), but only if the eight words from the scale–plane stack can be obtained in a single SSE2 load instruction. This requires a rearrangement of each row of the scale–plane stack (which is already done in the final part of the first phase). For this we define an array \text{shuffle}:

```cpp
for (int j = 0; j < w; j++)
    shuffle[j] = (j % stepAlpha) * nAlpha + j / stepAlpha;
```

Here, \( nAlpha \) corresponds to \( n_\alpha \), \( w \) to \( W \), and \( stepAlpha \) to \( s_\alpha \). This joins all columns having indices with the same remainder in a modulo operation with respect to \( s_\alpha \) in consecutive columns; see Fig. 7, bottom. All scale planes are thereby rearranged in a way that there are \( s_\alpha \) blocks of elements which, before rearrangement, had the same distance of \( s_\alpha \) from each other (Fig. 7, bottom). Note that all this presumes that the scale planes are stored such that \( j_\alpha \) varies within the rows of all scale planes (which are stored row–by–row).

Besides shuffling, the structure of the scale–plane stack has to be modified in another way: In each load operation, eight consecutive words are read from the same block of a scale plane. Since each scale plane is (double) cyclical, part of the vector of eight words could wrap around from the end of the block to the beginning. Rather than splitting the load operation, we fill in eight elements from the beginning of each block at its end.

An important difference to the serial CPU implementation is that no early termination is implemented: The summation of the minimal distances over all columns (along \( j_x \)) is not terminated as soon as the previously found minimum is exceeded. Since we sum eight values in parallel, early termination would probably be rather complex to realize. Early termination reduces the computation time in the serial implementation to about 60% — this advantage is lost in the parallel implementation. The search phase returns a match array containing the total distances for all visited \((j_\alpha, j_\psi)\) combinations, and the minimum search in this array is a separate step. Note that the summation of values uses saturated arithmetic (maximal value \text{FFFFH}).

Overall there are four nested loops in the search phase of the Min–Warping algorithm. The loop order has a considerable impact on the performance of the vectorized version as we found out in a pilot study (up to a speed factor of two on an Intel SandyBridge CPU). Among the best results were obtained by using \( j_x \) for the outer loop, followed by \( i_\psi, i_\alpha, \) and \( j_y \) (in this order; used for all reported experiments). This can be best explained by caching effects. To achieve thread parallelism on top of the vectorized implementation, we used OpenMP to parallelize the loop over \( i_\psi \). This solution
Figure 7. Shuffling of the scale–plane stack columns within each scale–plane row. Top: Original arrangement (numbers in different fonts) and access patterns (crosses x). Given eight consecutive values for $j_\alpha$ (with step width $s_\alpha$), the scale template is positioned at eight consecutive locations (five of which are depicted). In consequence, the scale–plane stack entries would not be accessed on consecutive addresses. Bottom: Shuffled arrangement. Here consecutive addresses are accessed.

was on the one hand convenient from a programming perspective, and on the other hand it actually reduces computation time by a factor slightly above the number of real CPU cores.

3.3. OpenCL Implementation for GPUs

The OpenCL implementation of the Min–Warping algorithm works in three distinct stages. In the first stage, the scale–plane stack is computed. The following search phase of Min–Warping (lines 2–33 in Fig. 6) is split into two stages (numbered two and three) to allow for easier parallelization. In the second stage, the minimum distance over $j_x$ is determined and stored in a memory buffer for every combination of $i_\alpha$, $i_\psi$, and $j_x$. Based on these stored values, in the third stage the total distance value over $j_x$ is accumulated for every combination of $i_\alpha$ and $i_\psi$, and finally the minimal total distance value is determined to identify the best parameter combination $(\hat{\alpha}, \hat{\psi})$. These three stages are described in the following in more detail. Each stage relies on a distinct compute kernel and corresponding wrapper function in host code; data structures and memory buffers in device and host memory are shared between stages. Fixed parameters like the scale factors $\sigma_k$ are located in constant device memory to ensure short access times. The work–group size was set to 256 work–items for all stages of the Min–Warping algorithm.

3.3.1. Stage 1: Computation of the scale–plane stack

The inputs to the Min–Warping algorithm are the panoramic images of the current view and snapshot with horizontal width $W$; they are transferred to device memory. Each pixel of these single–band images is represented by a 32–bit float number. This data format was chosen because GPUs are especially powerful in floating point computations. The computation of the scale–plane stack is a data–parallel problem which can be parallelized in a straightforward way by assigning a single work–item to every position $(j_\delta, j_\Theta)$ in the scale–plane
stack. All work–items are responsible for all scale factors $\sigma_k$ at their respective position and thus contribute to every scale plane. The scale planes themselves are also stored as 32–bit float numbers in device memory.

It is very important at this point that the device buffer containing the scale planes is organized in a special format which facilitates memory access in the subsequent search phase (Sect. 3.3.2). Each scale plane is divided into $s_\alpha$ partitions in memory with $s_\alpha = W/n_\alpha$ being the step size along the $j_\alpha$ direction; $l \in 0 \ldots s_\alpha - 1$ is the partition index. All partitions contain elements from all rows of the scale plane in row–major format (with row index $j_\delta$). However, each partition only contains part of the original columns of the scale plane (with column index $j_\Theta$). The assignment is such that the partition with index $l$ contains all columns from the original scale plane for which hold that $j_\Theta \mod s_\alpha = l$. This results in a memory arrangement as shown in Fig. 8.

3.3.2. Stage 2: First part of the search phase The most important inputs to the search phase are the scale–plane stack (already in global device memory from the preceding stage) and the scale template. The generated output is a buffer with $W \cdot n_\alpha \cdot n_\psi$ elements (Fig. 9) which contains for every position $(j_x, i_\alpha, i_\psi)$ in search space the minimum along the $j_y$ direction (column in the scale template and scale planes). Obviously, the computation of these minima is also a data–parallel problem: Each work–item can be assigned to a specific position $(j_x, i_\alpha, i_\psi)$ to search for the minimum along $j_y$ for this parameter combination. However, a naive implementation could easily result in very scattered memory accesses to the scale–plane stack because neighboring elements in the scale template usually index different scale planes $k$. Such scattered accesses to global device memory can severely degrade the performance of GPU implementations. For this reason, and because on a GPU the work–items within a work–group are scheduled together for execution, it is important that work–items within a work–group access contiguous regions in global memory at every memory operation.1

To achieve this goal, we have chosen a special arrangement of work–groups and work–items in the global OpenCL workspace (called “NDRange” in OpenCL nomenclature), combined with the

---

1This is a slight simplification: Even more important with regard to memory access and divergence are the subgroups of work–items within each work–group which are not only scheduled together, but really executed simultaneously or at least in immediate succession. These subgroups are called “warp” for Nvidia GPUs and “wavefront” for AMD GPUs.
Figure 9. Output buffer of the second stage. The memory arrangement corresponds to the OpenCL NDRange in which each work–item is responsible for one element of the output buffer. A single unidimensional work–group of the second stage (Sect. 3.3.2) is shown as bold rectangle. The work–group size of twelve work–items is only used for illustration purposes.

Figure 10. Illustration of the index conversion steps in the search phase from the work–item indices down to the $j_{\Theta}$ values which are used to address the elements $D[k][j_{\delta}][j_{\Theta}]$ in the scale planes. The step size $s_{\alpha}$ amounts to 2 in this example, the work–group size to 8. Note that the mapping between work–item indices and $i_{\alpha}$ can contain an additional offset (see Fig. 9).

special memory arrangement of the scale planes as described in the previous section. All work–items within a work–group share the same coordinate $j_{x}$, so that they work on the same column in the scale template and iterate through it along $j_{y}$ simultaneously.\(^2\) For this reason, all work–items retrieve the same scale–plane index $k$ from the scale template and therefore access the same scale plane.

\(^2\)As a positive side–effect, this avoids multiple code paths within a work–group, which can considerably decrease performance on GPUs.
What differs between the work–items within a work–group are the indices \( i_\alpha \) and \( i_\psi \), which are varied along one dimension of the NDRange such that consecutive work–items are assigned to consecutive values of \( i_\alpha \) (see Fig. 9). Depending on \( j_x \) and \( i_\alpha \), the index \( j_\Theta = j_x + j_\alpha = j_x + s_\alpha \cdot i_\alpha \) is addressed by a work–item within the selected scale plane \( k \). Because contiguous work–items are responsible for a contiguous range of \( i_\alpha \) values, the resulting combined memory access of the work–items fits perfectly to the memory layout as shown in Fig. 8 in which the \( j_\Theta \) indices are also arranged with the step size \( s_\alpha \) (see Fig. 10 for an illustration of the index conversion steps).

3.3.3. Stage 3: Final accumulation and minimum search

The goal of the final stage is to sum up the values over \( j_x \) for every combination \((i_\alpha, i_\psi)\) to the total distance values and to determine their global minimum. Therefore the output buffer of the second stage (in device memory) serves as input. The NDRange is unidimensional, and each work–item is assigned to a specific combination \((i_\alpha, i_\psi)\) to perform the accumulation work (in analogy to Fig. 9); the result of the accumulation is stored in local memory for fast access during the next computational step: After the accumulation, the work–items in each work–group search for the group–specific minimum according to a standard reduction scheme. Afterwards, the buffer which contains all the group–specific minima and corresponding \( i_\alpha \)– and \( i_\psi \)–indices is transferred to the host where the global minimum and the corresponding best match angles \((\hat{\alpha}, \hat{\psi})\) are determined.

3.4. Implementation for FPGAs

The target platform for the FPGA implementation is a PROCStar II PCI board provided by the company Gidel, which can also be used stand–alone. It contains four Altera Stratix-II EP2S60 FPGAs, each having access to two 64 MB DDR II SDRAMs. Only a single FPGA, including its associated SDRAMs, is used for the design.\(^3\) Three types of internal RAM are available within a single FPGA: 2 large blocks with 512 Kbit (MRAM), 255 intermediate blocks with 4 Kbit each (M4K), and 329 small blocks with 512 bit (M512). The host interface for data exchange using the PCI bus and the internal SDRAM controllers are provided by Gidel. A special component, the PROCMultiPort, allows access to the SDRAMs using multiple individual FIFO–buffered ports. A special software library enables user programs to access the PROCStar board. Data can be exchanged using registers, or transferred to the SDRAMs via DMA.

The actual design — developed in [36] based on an implementation of 2D–Warping described in [37, 38] — is also separated into two stages. The distance image computation array (DICA) computes the scale–plane stack, while the parameter search is done by the distance accumulation array (DAA). A central control unit and auxiliary components like the SDRAM controllers complete the design. During initialization, the current view and snapshot images are transferred from the host to one SDRAM each as 8–bit intensity values. The image width \( W \) must be a multiple of eight; also the image size cannot exceed \( 288 \times 64 \). The step sizes for \( \alpha \) and \( \psi \) are fixed to \( s_\alpha = s_\psi = 4 \), the number of scale planes is \( K = 9 \). While the scale–factor thresholds \( \tau_k \) can be changed in the control program, the scale factors \( \sigma_k \) are also fixed in the current implementation.

\(^3\)A single Stratix-II EP2S60 FPGA contains 24176 adaptive logic modules (ALM), 2.5 Mbit of internal RAM and 36 digital signal processing (DSP) blocks. The ALMs, consisting of two adaptive lookup tables (ALUT) and registers each, provide the actual logic capabilities, the DSPs can be used as multipliers.
3.4.1. Computation of the scale–plane stack  The individual scale planes are computed sequentially. As each entry only depends on a single pair of image columns from the snapshot and current view, multiple values can be computed in parallel. The DICA contains 64 distance image computation units (DICU) for this purpose, an MRAM memory block for the snapshot, and a multiplexer for writing back the results. Each DICU has a local memory buffer for the part of the current view processed by the unit.

Initially the snapshot data is copied from the SDRAM to the snapshot memory. The image data is rearranged during this transfer to simplify access to scaled image columns. The image is split into blocks of eight columns which are stored consecutively. Each eight–pixel row is stored as a single 64–bit data word. Also the image data of the current view is copied to local memory blocks, one M4K block storing 8–bit data words in each DICU. The data is distributed column–wise to the DICUs in a round–robin fashion, the first unit storing columns 0, 64, 128 etc. All image data is read sequentially from the SDRAMs for both snapshot and current view, while the data rearrangement is achieved by calculating the write addresses accordingly.

The actual computation of the distance values is rather straightforward: An outer loop iterates over the snapshot columns, while a second loop iterates over the blocks of 64 columns each in the current view. A third loop covers the rows inside the columns. Memory addresses, stored as fixed–point numbers to facilitate scaling of image columns, are used as loop variables. They are initialized with the correct address of the top row in the scaled image and incremented by a constant value — the scale factor (or its inverse for factors larger than one) represented as a fixed–point number. Only the integer part of the floating point address is used to access the memory blocks. As pixel values of eight consecutive columns are stored in each word of the snapshot memory, the correct value is selected in each step. The corresponding pixel values are read from the memory blocks and processed by the individual DICUs to compute the Euclidean distance between the image columns.

While the summation of squared pixel differences can be mapped readily to FPGA logic and digital signal processing (DSP) blocks, the final computation of the square root was implemented using the “Nonrestoring Binary Add–and–Subtract Square Rooting Algorithm” [39]. The resulting distance values are written back to the SDRAMs for later access during the search phase of the algorithm.\(^4\)

4The distance data in the scale planes has a width of 11 bits; this is sufficient to store the largest possible distance value, and this number of bits is used for processing during the search phase. However, for storage in the SDRAMs each 11–bit word is embedded into a 16–bit word.

3.4.2. Search phase  The data required during the search phase consists of the scale template as well as the entries from the scale–plane stack. While the scale template can be stored in local memory blocks, the considerably larger scale planes have to be read from the SDRAMs in which they are stored one by one in column–major format (referring to Fig. 3, left). As the SDRAMs can be accessed most efficiently using a burst–mode in which all data is read from consecutive addresses, it is advantageous to alter the search loops to iterate over \(j_\Theta\) and \(j_{\Theta'}\) instead of \(j_x\) and \(j_y\). While the outermost loop over \(i_\Psi\) is retained, different \(i_\alpha\) values can be processed in parallel by so called distance accumulation units (DAU). With \(i_\Psi\), \(i_\alpha\), \(j_\Theta\) and \(j_{\Theta'}\) given, the remaining parameters can be calculated as \(j_x = j_{\Theta'} - j_\Theta\), \(j_x = j_\Theta - s_\alpha i_\alpha\) and \(j_y = j_{\Theta'} + s_\Psi i_\Psi\), restricting them to the range 0 . . . \(W - 1\) by adding appropriate offsets if necessary. \(j_{\Theta'}\) is increased by four in each iteration, as multiple values are handled in each step (see below).
Each run over the scale–plane stack results in a complete run over the scale template, but most pairs of $j_x$ and $j_y$ access invalid entries (see Fig. 11). Additionally shifting $i_\alpha$ or $i_\psi$ by half the image width (denoted here by $i_\alpha'$, $i_\psi'$) results in four possible combinations, usually only one of these accessing a valid entry. Given the maximum image width of 288 pixels and the step sizes $s_\alpha = s_\psi = 4$, this allows to halve the search range for $i_\psi$ and $i_\alpha$ from 72 to 36, reducing the number of iterations and DAUs, respectively.

The scale template is stored in a compacted way to reduce memory requirements by moving the upper right triangle to the lower left (compare Fig. 11). As $s_\alpha$ is fixed, four consecutive columns can be stored in a single memory block. Also four template entries are combined into a single 16–bit data word. The whole template can be stored in 36 M4K memory blocks.

To increase parallelism each DAU contains four distance accumulation elements (DAE) that select the corresponding distance values from the scale–plane stack based on the values from the scale template for four consecutive values of $j_y$. Fig. 12 illustrates the data flow during the search phase for an image width of 24 pixels: Each DAU receives a data word (highlighted) from a template memory block at an address based on $j_y$ and the lowest bits of $j_\Theta$; the assignment between DAUs and memory blocks depends on $j_\Theta$, but is always consecutive and one-to-one. The four variables $dists$ each contain nine distance values of the scale–plane stack. In the example, the third DAE receives an invalid entry from the template and is ignored, while the others select the corresponding values from the $dists$ variables. The fourth DAE selects the shifted parameters $i_\alpha'$ and $i_\psi'$, as it

Figure 11. Two pairs of indices into the scale template, which both differ by half the image width (in this case 12), give four possible entries. In the general case, only one of these entries corresponds to a valid scale–plane index. See main text for details.
accesses the upper right triangle in Fig. 11 instead of the lower left. The DAU updates the minima for the selected combinations.

After each pass through $j_\Theta$ the minima are added to corresponding total distance values, and at the end of each iteration of $j_\psi$ the minimum total distance encountered so far is determined.

3.4.3. Control unit and software

The execution of the different phases of the Min–Warping algorithm is governed by a control unit containing a finite state machine (FSM). After receiving a start signal from the host computer, indicating that all required data (scale template and images) has been transferred to the SDRAMs, the FSM activates the different processing stages. First, the images are loaded into local memory buffers and the scale–plane stack is computed. The FSM provides the DICA with memory addresses and parameters. Afterwards the scale template is loaded into memory. For the search phase the FSM handles the loop variables and also suspends the computation when the data transfer from the SDRAMs stalls.

A control program running on the host computer is required to initialize the image data and start the computation. The images are loaded from files on disk and transferred to the FPGA board using DMA. Also the scale template is computed and transferred. Parameters like image size are set using special registers of the design accessible by the software.
Table II. Comparison of the characteristics of the different implementations

<table>
<thead>
<tr>
<th></th>
<th>CPU (serial)</th>
<th>CPU (SIMD)</th>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data types:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td>int32</td>
<td>int16</td>
<td>float32</td>
<td>int8</td>
</tr>
<tr>
<td>Scale plane</td>
<td>int32</td>
<td>int16</td>
<td>float32</td>
<td>int16</td>
</tr>
<tr>
<td><strong>Scale planes:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>$j_\theta$ vs. $j_\delta$</td>
<td>$j_\theta$ vs. $j_\delta'$</td>
<td>$j_\theta$ vs. $j_\delta$</td>
<td>$j_\theta$ vs. $j_\delta'$</td>
</tr>
<tr>
<td>Mem. org.</td>
<td>Unaltered</td>
<td>Row-wise shuffling and padding</td>
<td>Reordering in distinct partitions</td>
<td>Unaltered</td>
</tr>
<tr>
<td>Sqr. root</td>
<td>C function</td>
<td>Lookup table</td>
<td>OpenCL fct.</td>
<td>Special alg. [39]</td>
</tr>
<tr>
<td>Early term. during search</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

3.4.4. Design sizes and variations  
Because the compiled design requires most of the resources on the Stratix-II (about 98% of the adaptive lookup tables [ALUT] and/or registers are used), the image width had to be fixed to 288 as maximum. However, more recent FPGAs offer considerably more computational and memory resources and would presumably allow for larger image widths.

To verify this claim and to check for manufacturer-specific differences, we compiled our FPGA design also for a Xilinx Virtex-7 485T as target. The Xilinx Virtex-7 family was introduced about five years after the Altera Stratix-II and is still recommended for new designs in 2015. Thus, it is better comparable to the CPUs and GPUs used in the other warping implementations.

On the Altera Stratix-II 60, the warping core of the design (without the manufacturer-specific SDRAM- and PCI-parts) occupies about 73% of the ALUTs (34,969 of 48,352), 24% of the DSP 9x9 multipliers (68 of 288), and 37% of the Block Memory bits (933,888 of 2,544,192). In contrast, the same warping core needs only about 23% of the logic elements called “slice” (17,065 out of 75,900) and 2% of the DSP48E1 blocks (66 out of 2,800) on the Xilinx Virtex-7 485T. For the about 1 Mbit of memory, none of the 1,030 RAMB36E1 blocks (36 Mbit) but 14,889 slice LUTs are used. In conclusion, the Virtex-7 would definitely allow for a scaled-up version of the basic FPGA design for considerably larger image sizes. However, for our experiments only a Stratix-II was available.

3.5. Implementations: Summary

Table II lists the main differences between the implementations. Most differences are imposed by the underlying hardware and the corresponding optimizations, e.g. early termination in search is restricted to the serial CPU implementation. Data types are chosen differently between platforms because there is no single data type which would yield optimum performance on all platforms.

In summary, the most important optimization technique used for the CPU–SIMD implementation is the careful design of memory access patterns and loop orders to make best use of the CPU caches. Furthermore, memory contents are rearranged for efficient SIMD load operations. In the GPU implementation, the rearrangement of memory contents plays also an important role — here with the purpose of allowing coalesced access to global memory from the work–items in a work–group. In addition, divergence between work–items in a work–group is avoided and local memory
Figure 13. Home vector field for all $10 \times 17$ grid positions of the used image database. The snapshot is marked by a red circle, all other locations serve as current views (parameter settings: $W = 288$, $K = 9$, $n_{\alpha} = n_{\psi} = 72$).

Figure 14. Comparison regarding the number of scale planes ($K$) for an image width $W$ of 288 pixels. The graphs show the median of the angular error over varying numbers of $\alpha$– and $\psi$–steps ($n_{\alpha}, n_{\psi}$). For details see text.

is used to speed up reductions. In the FPGA implementation, the opposite approach was chosen with regard to memory handling: The scale–plane data is preserved in its original order, while the algorithmic structure is adapted to ensure sequential memory access. The FPGA allows for full control in every clock cycle, and much care has to be taken to keep all computation units fully utilized in every step.

4. EXPERIMENTAL STUDIES AND RESULTS

4.1. Warping Quality

There are four parameters in the Min–Warping algorithm which directly influence the problem size and therefore the computational demands. These are the number of scale planes $K$, the number of $\alpha$– and $\psi$–steps ($n_{\alpha}, n_{\psi}$), and the width $W$ of the panoramic images. We first examine how these
parameters influence the resulting warping quality because an increase of the problem size is only justified if the results get better. In a previous study [2], several quality measures were introduced for the comparison between homing algorithms. From these measures, we here use the median of the angular error, which is computed from a unit–length true home vector $h$ and an estimated home vector $\hat{h}$ by

$$\epsilon = \arccos(h^T\hat{h}) \in [0, \pi].$$

(2)

Fig. 13 shows the generated home vectors for a single snapshot position close to the center of the room. Overall the precision is very good, however, it is noticeable that the deviation between the generated home vectors and the ideal homing direction increases with the distance between snapshot and current view. For quality assessment, all possible combinations of snapshot and current view were tested, resulting in $170 \times 169$ angular errors $\epsilon$, the median of which is the final measure of homing quality for a specific parameter combination of the Min–Warping algorithm.

In Fig. 14 the median of the angular error (MAE) is reported for several parameter combinations (computed with the OpenCL implementation). All graphs show that the MAE generally decreases the larger the number of $\alpha$ and $\psi$ steps ($n_\alpha$, $n_\psi$) is. Thus, $n_\alpha = n_\psi = 288$ is clearly preferable compared to $n_\alpha = n_\psi = 72$. With regard to the number of scale planes, $K = 13$ mostly yields the best results and $K = 9$ the worst. However, especially for $n_\alpha = n_\psi = 288$ the setting $K = 17$ offers a tiny advantage over $K = 13$ and results in the overall lowermost MAE. An increase in the image width $W$ does not improve the homing quality any further by a significant amount (data not shown). In conclusion, for most precise results the parameter combination $W = 288$, $K = 17$, $n_\alpha = n_\psi = 288$ should be chosen, and for the smallest problem size $W = 288$, $K = 9$, $n_\alpha = n_\psi = 72$. Although an image width of $W = 1152$ offers no advantage in practice, we included it into the following speed tests because it allows for the largest problem sizes within the tested parameter range.

4.2. Performance

4.2.1. Results on the FPGA The execution time of the FPGA design (for the device specifications see Sect. 3.4) was measured by counting clock cycles. Dividing the number of cycles by the clock
Table IV. Device specifications (CPUs and GPUs)

<table>
<thead>
<tr>
<th>Compute device</th>
<th>Clock [GHz]</th>
<th># of shaders</th>
<th># of cores/hardware threads</th>
<th>$R_{\text{peak}}^{\text{SP}}$ [GFlops]</th>
<th>$R_{\text{peak}}^{\text{int16}}$ [GIops]</th>
<th>TDP [W]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom N2600</td>
<td>1.6</td>
<td>—</td>
<td>2/4</td>
<td>38.4</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>Core i7-2600</td>
<td>3.5</td>
<td>—</td>
<td>4/8</td>
<td>336</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>NVS4200M</td>
<td>1.62</td>
<td>48</td>
<td>—</td>
<td>155</td>
<td>—</td>
<td>&lt; 20</td>
</tr>
<tr>
<td>GeForce GTX570</td>
<td>1.4</td>
<td>480</td>
<td>—</td>
<td>1344</td>
<td>—</td>
<td>219</td>
</tr>
<tr>
<td>Radeon HD7970</td>
<td>1.05</td>
<td>2048</td>
<td>—</td>
<td>4301</td>
<td>—</td>
<td>250</td>
</tr>
</tbody>
</table>

Notes: $R_{\text{peak}}^{\text{SP}}$ values are given for operations on 32–bit floating point numbers, $R_{\text{peak}}^{\text{int16}}$ values for operations on 16–bit integer values (mix of additions and multiplications: GIops stands for “billion integer operations per second”). The $R_{\text{peak}}^{\text{int16}}$ values are derived from the specifications provided in [5, 41].

frequency gives the runtime in milliseconds. An additional counter measured the delays caused by memory transfers (stalls). The time required for the DMA transfer was measured using a (monotonic) system clock. After starting the transfer a function provided by the Gidel library was used to wait for the end of the transfer.

For our testing, the FPGA was run at a clock rate of 50 MHz. The execution time was measured for all possible combinations of 170 snapshots and 170 current views for the parameter settings $W = 288$, $K = 9$, $n_\alpha = n_\psi = 72$. As the number of execution cycles required for the processing stages is constant for a given set of parameters, only the number of stalls can vary. The results are given in Table III. The total values include all necessary operations to compute a single home vector from a CV/SS pair.

The stalls are caused during the search phase by the limited memory bandwidth of the two SDRAMs which are connected to the FPGA. These memory blocks could be replaced by more but smaller blocks when for example building an ASIC based on the design. Also, a more recent FPGA type with more internal memory could store the scale planes internally with no SDRAM access at all. For this reason, we think it is acceptable to disregard these additional delays when comparing the speedup further below.

In our comparison between the Altera Stratix-II and the Xilinx Virtex-7 FPGA family (see Sect. 3.4.4) no qualitative differences in the implementation of the Min–Warping algorithm could be seen. All needed element types in one family have their counterpart in the other family, thus Min–Warping could be implemented on Altera and on Xilinx devices without changes in the basic implementation. However, the age of an FPGA family has a strong impact on the number of available resources and therefore on the possible degree of parallelization. Here one would expect a much better performance from the newer Virtex-7 FPGA family.

4.2.2. Results on CPUs and GPUs As described in Sect. 3, there are two CPU implementations (serial and multi–core–SIMD; the latter will be just denoted as SIMD in the following) and one GPU implementation (in OpenCL) of the Min–Warping algorithm. These were run on several devices from two classes. In the class of “low–power” processors we included the Intel Atom N2600 as CPU and the Nvidia NVS4200M as GPU, in the class of “high–end” processors the Intel Core
### Table V. Execution time [ms]: SPS generation

<table>
<thead>
<tr>
<th>Configuration (W/K)</th>
<th>Low–Power Processors</th>
<th>Processor and implementation</th>
<th>N2600 (CPU), Serial</th>
<th>N2600 (CPU), SIMD</th>
<th>NVS4200M (mob. GPU), OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>288/9</td>
<td>166 (1.5)</td>
<td>19 (0.43)</td>
<td>5.2 (0.024)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/13</td>
<td>240 (0.79)</td>
<td>27 (0.33)</td>
<td>7.4 (0.037)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/17</td>
<td>313 (1.2)</td>
<td>35 (0.65)</td>
<td>9.6 (0.039)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/9</td>
<td>5913 (3.1)</td>
<td>1027 (5.8)</td>
<td>398 (0.14)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/17</td>
<td>12746 (15)</td>
<td>1923 (5.4)</td>
<td>762 (0.25)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:** N = 169 except for “N2600 (CPU), Serial” combined with “1152/17” (N = 20); standard deviations given in parentheses.

### Table VI. Execution time [ms]: Search phase

<table>
<thead>
<tr>
<th>Configuration (W/K/n_a)</th>
<th>Low–Power Processors</th>
<th>Processor and implementation</th>
<th>N2600 (CPU), Serial</th>
<th>N2600 (CPU), SIMD</th>
<th>NVS4200M (mob. GPU), OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>288/9/72</td>
<td>1186 (288)</td>
<td>102 (0.81)</td>
<td>28 (0.047)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/9/144</td>
<td>4680 (1142)</td>
<td>356 (5.1)</td>
<td>106 (0.095)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/9/288</td>
<td>17138 (4335)</td>
<td>1390 (4.0)</td>
<td>478 (0.56)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/17/72</td>
<td>2056 (528)</td>
<td>115 (0.90)</td>
<td>30 (0.10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/17/144</td>
<td>8020 (2105)</td>
<td>395 (1.4)</td>
<td>110 (0.081)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/17/288</td>
<td>31737 (8660)</td>
<td>1469 (8.9)</td>
<td>477 (0.66)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/9/72</td>
<td>37071 (7568)</td>
<td>2992 (109)</td>
<td>602 (0.61)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/9/144</td>
<td>153022 (31334)</td>
<td>8877 (103)</td>
<td>1995 (2.4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/9/288</td>
<td>605645 (124398)</td>
<td>34383 (136)</td>
<td>7984 (2.4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/17/72</td>
<td>53318 (1832)</td>
<td>3295 (140)</td>
<td>660 (0.54)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/17/144</td>
<td>211757 (7289)</td>
<td>9396 (83)</td>
<td>2033 (2.4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/17/288</td>
<td>812342 (28091)</td>
<td>35445 (100)</td>
<td>7956 (1.5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:** N = 169 except for “N2600 (CPU), Serial” combined with “1152/17/*” (N = 20); standard deviations given in parentheses.

i7-2600 as CPU and the AMD Radeon HD7970 and the Nvidia Geforce GTX570 as GPUs. In this way, we have a large spectrum of devices with different microarchitectures covered. Table IV gives an overview of the specifications and theoretical peak performance numbers of the CPU and
Figure 15. Speedup relative to the serial implementation run on the Core i7-2600; values based on the combined entries (SPS generation plus search phase) in Tables VII and VIII.

Table VII. Execution time [ms]: SPS generation

<table>
<thead>
<tr>
<th>Configuration (W/K)</th>
<th>High-End Processors</th>
<th>Processor and implementation</th>
<th>i7-2600 (CPU), Serial</th>
<th>i7-2600 (CPU), SIMD</th>
<th>GTX570 (GPU), OpenCL</th>
<th>HD7970 (GPU), OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>288/9</td>
<td>16 (1.2)</td>
<td>2.3 (0.12)</td>
<td>0.75 (0.057)</td>
<td>1.0 (0.069)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/13</td>
<td>23 (1.3)</td>
<td>3.2 (0.070)</td>
<td>1.00 (0.020)</td>
<td>1.2 (0.084)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>288/17</td>
<td>30 (0.91)</td>
<td>4.3 (0.17)</td>
<td>1.2 (0.014)</td>
<td>1.3 (0.052)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/9</td>
<td>789 (0.58)</td>
<td>82 (0.82)</td>
<td>31 (0.17)</td>
<td>14 (0.29)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152/17</td>
<td>1489 (0.54)</td>
<td>155 (0.34)</td>
<td>59 (0.32)</td>
<td>26 (0.22)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: $N = 169$; standard deviations given in parentheses.

GPU devices. Depending on the mainly used data type in our implementations (CPU–SIMD: 16-bit integer; GPU: 32-bit float), either the theoretical integer or floating point peak performance is listed.

For each implementation on each device, the execution time for a specific parameter combination was measured over 170 pairs of snapshot and current view. The first trial was excluded because it sometimes contains initialization costs, thus the average execution time and its standard deviation...
Table VIII. Execution time [ms]: Search phase

<table>
<thead>
<tr>
<th>Configuration (W/K/n_α)</th>
<th>i7-2600 (CPU), Serial</th>
<th>i7-2600 (CPU), SIMD</th>
<th>GTX570 (GPU), OpenCL</th>
<th>HD7970 (GPU), OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>288/9/72</td>
<td>100 (21)</td>
<td>5.6 (0.18)</td>
<td>2.8 (0.034)</td>
<td>2.1 (0.31)</td>
</tr>
<tr>
<td>288/9/144</td>
<td>395 (86)</td>
<td>19 (0.30)</td>
<td>9.9 (0.038)</td>
<td>5.0 (0.31)</td>
</tr>
<tr>
<td>288/9/288</td>
<td>1573 (345)</td>
<td>73 (0.60)</td>
<td>38 (0.073)</td>
<td>16 (0.14)</td>
</tr>
<tr>
<td>288/17/72</td>
<td>113 (26)</td>
<td>6.0 (0.19)</td>
<td>3.0 (0.061)</td>
<td>2.2 (0.18)</td>
</tr>
<tr>
<td>288/17/144</td>
<td>442 (101)</td>
<td>20 (0.26)</td>
<td>9.9 (0.093)</td>
<td>5.3 (0.054)</td>
</tr>
<tr>
<td>288/17/288</td>
<td>1838 (419)</td>
<td>73 (0.45)</td>
<td>38 (0.12)</td>
<td>17 (0.29)</td>
</tr>
<tr>
<td>1152/9/72</td>
<td>2898 (589)</td>
<td>269 (1.7)</td>
<td>50 (0.12)</td>
<td>28 (0.035)</td>
</tr>
<tr>
<td>1152/9/144</td>
<td>10570 (2160)</td>
<td>692 (0.71)</td>
<td>147 (0.46)</td>
<td>106 (0.39)</td>
</tr>
<tr>
<td>1152/9/288</td>
<td>41070 (8420)</td>
<td>2453 (1.9)</td>
<td>562 (1.2)</td>
<td>350 (0.22)</td>
</tr>
<tr>
<td>1152/17/72</td>
<td>3477 (746)</td>
<td>282 (0.79)</td>
<td>50 (0.098)</td>
<td>30 (0.022)</td>
</tr>
<tr>
<td>1152/17/144</td>
<td>12270 (2655)</td>
<td>797 (0.79)</td>
<td>149 (0.41)</td>
<td>112 (0.14)</td>
</tr>
<tr>
<td>1152/17/288</td>
<td>47140 (10256)</td>
<td>2620 (25)</td>
<td>564 (1.0)</td>
<td>364 (1.1)</td>
</tr>
</tbody>
</table>

Notes: N = 169; standard deviations given in parentheses.

Figure 16. Speedup relative to the serial implementation run on the Atom N2600; values based on the combined entries (SPS generation plus search phase) in Tables V and VI. The value for the FPGA is from the configuration “fast, 11 bit, 50 Hz” (total time excl. stall; see Table III).

were determined from a sample size of N = 169. Time measurements were split into the time required for SPS generation and for the search phase.

Low–power processors Looking at the results for SPS generation on low–power processors (Table V), the SIMD implementation achieves a speedup between 6.6 and 9 compared to the serial implementation on the Atom N2600. For the search phase (Table VI), the speedup varies between
12 (smallest problem size) and 23 (largest problem size). This exceeds even the expected range for a dual-core CPU with eightfold SIMD parallelism.

Comparing SIMD on the N2600 with OpenCL on the mobile GPU NVS 4200M for SPS generation (Table V), the additional speedup varies between 3.6 for an image width of $W = 288$ and 2.5 for $W = 1152$. In the search phase (Table VI), the results are the other way round: a speedup of about 3.5 for $W = 288$ and up to 5 for $W = 1152$. Overall, the mobile GPU outperforms the Atom CPU by a considerable margin.

Looking at the overall warping process, i.e. SPS generation plus search phase to compute a single home vector from a CV/SS pair, the achieved speedup values for low-power processors (compared to the serial implementation on the N2600) vary between 10.3 (SIMD for parameter combination 1152/9/72; notation: $W/K/n_\alpha (= n_\psi)$) and 94.9 (GPU NVS4200M for 1152/17/288) (Fig. 16). At the smallest problem size (288/9/72), the speedup value for SIMD is 11.2, for the NVS4200M 40.1, and for the FPGA 42.9. Thus, the FPGA in its fastest configuration beats all other approaches. For the problem size with most precise results (288/17/288), the SIMD speedup amounts to 21.3 and the GPU speedup to 65.9. For the largest problem size (1152/17/288), this difference is even larger: speedup of 22.4 for SIMD and of 94.9 for GPU.

### High-end processors

For the high-end CPU Core i7-2600 the achieved speedup between the serial and the SIMD implementation varies between 7 and 9.6 for SPS generation — the latter value for the largest problem sizes (Table VII). In the search phase, the speedup peaks at 25 for the parameter combination 288/17/288 and goes down to values as low as 11 for the combination 1152/9/72 (Table VIII). This is quite the opposite behavior as on the Atom CPU; furthermore, these values stay below the expected maximum speedup of 32 (quad core with eightfold SIMD parallelism).

Moving forward to the high-end GPUs, we first notice that the GTX570 achieves a speedup between 3.6 and 2.6 for SPS generation (Table VII), compared to SIMD on the i7-2600. Interestingly, larger speedup values are found for smaller problem sizes. The HD7970 shows the opposite behavior: A speedup of 2.3 for the smallest problem size and of ca. 6 for the largest problem size. In the search phase (Table VIII), the results are more consistent between the GPUs. For the GTX570, the speedup compared to SIMD on the i7-2600 varies between 2 for $W = 288$ and up to 5.6 for $W = 1152$. For the HD7970, the range is between 2.7 for the smallest problem size and values up to 9.6 for $W = 1152$.

The speedup for the overall warping process on high-end processors (compared to the serial implementation on the i7-2600) varies between 10.4 (SIMD for parameter combination 1152/9/72) and 125 (GPU HD7970 for 1152/17/288) (Fig. 15). At the smallest problem size (288/9/72), the speedup value for SIMD is 15.0, for the GTX570 33.2, and for the HD7970 37.8. For the problem size with most precise results (288/17/288), the values are 24.2 (SIMD), 47.4 (GTX570), and 102 (HD7970). At this point the multi-core-SIMD implementation is clearly outperformed by the HD7970 (less so by the GTX570). For the largest problem size (1152/17/288) the speedup values amount to 17.5 (SIMD), 78.2 (GTX570), and 125 (HD7970), thus the performance advantage of the GPUs is even stronger. This is further corroborated by directly comparing SIMD to GPUs: The speedup achieved by the GTX570 (vs. SIMD) is in the range between 2.0 (for the parameter...
combination 288/9/288) and 4.6 (for 1152/17/144), whereas the speedup values for the HD7970 vary between 2.6 (for the smallest problem size: 288/9/72) and 8.4 (for 1152/9/72).

4.3. Power Efficiency

Power efficiency is an important aspect in the evaluation of computer hardware and software implementations. This is especially true for the Min–Warping algorithm which is intended to run on mobile robots with limited battery capacity. However, also in the area of high–end processors power consumption is a limiting factor because of the corresponding costs and thermal problems (e.g., for supercomputers), and Min–Warping can serve here as an exemplary algorithm which contains many common operations like the computation of scalar products and Euclidean distances, accumulation, and search.

We measured the power input for the following parameter combinations: 288/9/72, 288/9/144, 288/9/288, 1152/17/72, 1152/17/144, 1152/17/288. In this way we covered both the smallest and the largest problem size. Measurements were carried out for the SIMD and the OpenCL implementation on all applicable devices and for the FPGA setup. Furthermore, we measured the power input at idle for each setup (the high–end GPUs and the FPGA card were removed from the PCs for this measurement; however, this was not possible for the NVS4200M because it is built into a notebook). All measurements were carried out at the AC input of each power supply with a high–precision multimeter (METRAHit 29S) and recorded over a serial connection with a Windows PC and the application METRaWin. Concerning the notebooks in the study (Atom N2600, NVS4200M), the batteries were removed before and the screens turned off. Every power measurement extended over 100 seconds with a sampling interval of one second. The mean value of this sample with size $N = 100$ is the final metered value.

Because we are interested in the real power consumption of the devices, all values which are reported in the following are corrected for the power efficiency of the respective power supplies at the respective load. For the notebook power supplies, we determined the efficiency in our own laboratory for the relevant output range, for the power supplies in the desktop PCs (Enermax Revolution 85+ 1250W for the HD7970, Enermax ErPRO80+ 500W for the GTX570 and i7-2600, Delta Electronics FS216U300PCW for the FPGA) we relied on efficiency measurements by external sources\(^7\) or estimated the efficiency according to typical findings from the time of their market launch\(^8\).

Furthermore, we ensured that CPU activity was at the required minimum for the measurements under load with GPUs and FPGA. This was not trivial for the NVIDIA GPUs (GTX570 and NVS4200M) because their OpenCL implementation relies on active waiting, i.e. a thread on the CPU is polling in a close loop if the operations on the GPU are finished. To overcome this problem, we followed the suggestion by [3] and replaced the `sched_yield` function from the GNU C library by our own implementation which calls `usleep` instead.

Figure 17 shows the relative energy consumption for all parameter combinations for a single run of Min–Warping;\(^9\) “relative” means here that the power draw at idle is subtracted from the power consumption.

---

\(^7\)For Enermax Revolution 85+ 1250W see \[42\], for Enermax ErPRO80+ 500W see \[43\].
\(^8\)For Delta Electronics FS216U300PCW see \[44\].
\(^9\)Energy consumption is computed as the product of the average required time for a single run of Min–Warping and the measured and corrected power input.
Figure 17. Energy consumption of the SIMD and OpenCL implementations (SPS generation and search phase for a single SS/CV pair). Please note the logarithmic scale. The maximum relative standard error is smaller than 0.9%. The given values are relative to the power draw at idle of each computer setup (in case of the FPGA and the graphics adapters GTX570 and HD7970, the expansion cards were completely removed from the PCs for the measurement at idle). The value for the FPGA is based on the total execution time excl. stall (Table III).

draw at load before the energy consumption is computed. This relative view considers only the energy consumption which is directly caused by executing the algorithm. At the smallest problem size, the SIMD implementation on the Atom N2600 is the most efficient one (0.27 Ws), closely followed by the FPGA (0.31 Ws). Surprisingly, the mobile GPU NVS4200M comes in last (0.82 Ws) while the high-end GPU HD7970 fairs pretty well (0.40 Ws). With increasing problem size, the picture changes in favor of the high-end GPUs, especially the HD7970. From the parameter combination 1152/17/72 on it has the smallest energy consumption (10.6 Ws compared to 14.4 Ws required by the N2600). Further notable observations are that the NVS4200M has the highest energy demands throughout the whole parameter range, the SIMD implementation on the Core i7-2600 is usually somewhere in the middle, and the GTX570 is less efficient than the HD7970 (by a factor between 1.5 and 2).

Another valid view on energy consumption is to base the reported values on the power draw at load without substracting the power draw at idle. The resulting “absolute” values reflect the fact that one always needs a whole computer system to run an algorithm; only the FPGA is excluded from this kind of comparison because the FPGA card can also be used in stand-alone mode without being built into a PC. Fig. 18 for the absolute energy consumption shows a similar picture as Fig. 17 with the relative values. However, the high-end GPUs profit from the absolute view because their power draw at load (e.g., GTX570 at largest problem size: 230W) strongly exceeds the power draw at idle of a modern desktop PC (our measurement for the computer system which is used for the GTX570: 34W). For this reason the HD7970 is now consistently the most efficient device, even for the smallest problem size, and the GTX570 surpasses SIMD on the i7-2600 more clearly at large problem sizes. At small problem sizes, however, SIMD on the i7-2600 is still more efficient than the GTX570. In numbers: Compared to SIMD on the i7-2600, the energy consumption is reduced by the GTX570 by a factor up to 1.7 (for the parameter combination 1152/17/144), and by the HD7970...
Figure 18. Absolute energy consumption of the SIMD and OpenCL implementations (SPS generation and search phase for a single SS/CV pair). Please note the logarithmic scale. The maximum relative standard error is smaller than 0.9%. The presented values for the HD7970 were corrected downward to account for the difference in power consumption at idle between the PC with the GTX570 and the HD7970.

by a factor up to 3.0 (for 1152/17/72). Moreover — looking again at the low–power processors — under the absolute view the NVS4200M performs less badly compared to the Atom N2600 than under the relative view.

5. DISCUSSION

We examined in our study how a single–threaded and scalar implementation of the Min–Warping algorithm can be accelerated by either a multi–core–SIMD approach on CPUs, an OpenCL implementation on GPUs, or by putting it onto an FPGA. Furthermore, we analyzed the resulting energy consumption.

5.1. Low–power processors and real time operation

For low–power processors, we found that multi–threading and vectorization gave the CPU Atom N2600 a considerable boost so that it appears to be the most energy–efficient platform for small images (which would be used in practice on a mobile robot). However, the FPGA requires only slightly more energy while offering a much better speedup (42.9 vs. 11.2; see Fig. 16). Furthermore, the relative energy consumption of the FPGA in Fig. 17 may be interpreted as absolute value because the FPGA board can be used in stand–alone mode without a PC. In this case the absolute energy consumption of the FPGA would only amount to half of the energy consumption of the Atom CPU (see Fig. 18). In addition, the FPGA used in these studies is about five years older than the CPU. The more recent FPGA generations show a better energy efficiency, and there are even specialized low–power FPGA families available while we used a device of a mid to high performance family. In conclusion, the best combination of energy consumption and speed is definitely offered by the FPGA approach.
If we increase the problem size to achieve more precise results, our specific FPGA setup cannot cope with the stronger hardware demands and the Atom N2600 would be the best choice for a battery–limited mobile platform, because the mobile GPU NVS4200M — although considerably faster than multi–core–SIMD on the Atom CPU — falls short regarding energy consumption. An FPGA–based implementation for a larger problem size is possible but would need another board with either an FPGA with more internal memory capacity or higher external memory throughput, i.e. more and faster DDR RAM channels connected to the FPGA. A more recent FPGA (like the Virtex-7 which was addressed in Sect. 3.4.4) could be used for this purpose. However, when comparing CPU or GPU results with the FPGA–based implementations it should be noted that large FPGA devices are more expensive than the CPUs and GPUs used here.

With regard to real–time operation, the FPGA is definitely usable for this purpose. It would achieve for the smallest problem size more than 30 home vector computations per second (see the total computation time excl. stall in Table III) and therefore enable the smooth update of the movement direction of a mobile robot in a visual navigation task. The same is true for the mobile GPU and with some restrictions also for the Atom CPU. With the latter one, one can achieve a home vector update rate of ca. 8 Hz at the smallest problem size. This is already sufficient for smooth movement trajectories, assuming a moderate velocity of the mobile robot. Moreover, additional heuristics exist which accelerate the home–vector computation by restricting the search space in the second phase of the Min–Warping algorithm [2]. These heuristics could be used to achieve real–time performance on the Atom CPU also for larger problem sizes and more precise home vector computations.

5.2. High–end processors

When we include the high–end processors in the comparison, the picture changes in favor of GPUs. The HD7970 is (nearly) consistently the fastest device and at the same time the most energy–efficient one, at least regarding the absolute energy consumption (Fig. 18). At large image sizes the HD7970 surpasses the Atom N2600 also with regard to the relative energy consumption. The speed advantage of the HD7960 compared to the GTX570 amounts to a factor of about two and is therefore smaller than one would expect from the theoretical maximum floating point performance $R_{\text{peak}}$ (HD7970: 4301 GFLops; GTX570: 1344 GFlops; see Table IV). Nevertheless, because both GPUs have a similar power draw, the HD7970 is the more efficient device. Multi–core–SIMD on the high–end CPU Core i7-2600 is clearly surpassed by both high–end GPUs, especially at large problem sizes.

However, our results confirm that the common comparison between serial CPU implementations and GPU implementations is quite misleading (e.g., [9, 11, 13]): In such scenarios very large speedups in favor of GPUs are achieved; however, the GPU advantage disappears either completely as soon as the full CPU capacities are utilized [16, 3, 23, 15] or becomes at least considerably smaller [7, 20, 32] (speedups in the single–digit range). This is what we observe for the Min–Warping algorithm as well with speedups between 2 and 8.4 in favor of GPUs compared to multi–core–SIMD. Nevertheless, even with these rather small speedup–numbers at least high–end GPUs offer the more energy–efficient solution with energy savings by a factor up to 3. This is consistent with the findings by Scogland et al. [8] and Tian et al. [19].
Because the required number of computations in the Min–Warping algorithm increases linearly with respect to both the number of $\alpha$ and $\psi$ steps, for $n = n_\alpha = n_\psi$, the complexity is $O(n^2)$. This is closely reflected by the observed execution times of the serial CPU implementation in the search phase (Tables VI and VIII). However, in the parallel implementations the execution times increase with less than quadratic progression. On the high–end GPUs, this can be explained by the number of work–items which are created in the search phase: This number is equal to $n_\alpha \cdot n_\psi$. For $n_\alpha = n_\psi = 72$, this amounts to 5184 work–items which is well below the maximum number of threads in flight\textsuperscript{10} (GTX570: 23040); therefore the GPUs are underutilized and hiding long–latency operations like memory accesses by switching to other sub–groups of work–items (Nvidia: warp; AMD: wavefront) does not work well. When $n_\alpha$ and $n_\psi$ increase, the GTX570 and HD7970 get more efficient at latency hiding, which speeds up execution.

6. CONCLUSIONS

In conclusion, our findings support the view that for highly parallel workloads and appropriately large problem sizes modern high–end GPUs are both faster and more energy–efficient than classical CPUs, even if the latter are fully utilized by employing multi–core and SIMD programming. When we move to the area of low–power devices, the FPGA in our study really stood out by simultaneously being fast and highly energy–efficient. The Atom CPU in combination with multi–core–SIMD programming is recommendable if more flexibility is required; real–time Min–Warping on mobile robots is possible with this processor.

ACKNOWLEDGEMENT

We would like to thank Klaus Kulitza from Bielefeld University for measuring the power efficiency of the notebook power supplies and for his general assistance in carrying out the power measurements. Furthermore, we appreciate that Andreas Kolb and Martin Lambers from the University of Siegen provided us with first pointers to the relevant literature.

REFERENCES


\textsuperscript{10}“Threads in flight” have their contexts stored on GPU registers and are ready to start execution once the required computational resources are available or data requested from memory has been delivered.


