J Real-Time Image Proc
DOI 10.1007/s11554-013-0390-x

ORIGINAL RESEARCH PAPER

Accelerated catadioptric omnidirectional view image unwrapping processing using GPU parallelisation

Nguan Soon Chong · M. L. Dennis Wong · Yau Hee Kho

Received: 14 November 2013 / Revised: 8 December 2013 / Accepted: 13 December 2013

Abstract Catadioptric omnidirectional view sensors have found increasing adoption in various robotic and surveillance applications due to their 360° field of view. However, the inherent distortion caused by the sensors prevents their direct utilisations using existing image processing techniques developed for perspective images. Therefore, a correction processing known as "unwrapping" is commonly performed. However, the unwrapping process incurs additional computational loads on central processing units. In this paper, a method to reduce this burden in the computation is investigated by exploiting the parallelism of graphical processing units (GPUs) based on the Compute Unified Device Architecture (CUDA). More specifically, we first introduce a general approach of parallelisation to the said process. Then, a series of adaptations to the CUDA platform is proposed to enable an optimised usage of the hardware platform. Finally, the performances of the unwrapping function were evaluated on a high-end and low-end GPU to demonstrate the effectiveness of the parallelisation approach.

Keywords Omnidirectional sensor · Image unwrapping · GPU · Parallelisation · CUDA · Bilinear interpolation

N. S. Chong (✉) · M. L. D. Wong
Faculty of Engineering, Computing and Science,
Swinburne University of Technology (Sarawak Campus),
93350 Kuching, Sarawak, Malaysia
e-mail: nschong@swinburne.edu.my

Y. H. Kho
School of Engineering, Nazarbayev University,
Astana, Kazakhstan
e-mail: yauhee.kho@nu.edu.kz

1 Introduction

Over the past decades, catadioptric omnidirectional view sensors (COVS) have gained increasing popularity in particular in robotic and surveillance applications. This is mainly attributed to the large field of view provided by the sensors thus allowing simultaneous monitoring of the surrounding in different view angles. For most robotic applications, however, the main difficulty in utilising omnidirectional view images has been the inherent view distortion caused by the curvature of the sensors. This distortion causes a problem when conventional computer vision techniques, such as those related to recognition and detection, are applied to the system. In order to remove the distortion and correct the omnidirectional view images into perspective view, one would require that the captured image be unwrapped.

Several unwrapping techniques had been previously developed to cater to different mirror profiles and applications. For COVS of single viewpoint (SVP) [1], Lei et al. [12] developed a common unified panoramic unwrapping method that utilises Scaramuzza et al.'s calibration technique [19]. Their approach relies on a Taylor series approximation and checker-box patterns distorted by the mirror's curvature to calibrate the overall sensor. Nayar [14] also previously proposed an unwrapping method for paraboloidal mirrors using geometrical solution. For non-SVP mirrors such as the spherical ones, Jeng and Tsai [11] proposed a calibration using ground-truth information to achieve panoramic unwrapping on all mirror profiles. Other noteworthy works also include the ground plane unwrapping method proposed by Hicks and Bajcsy [9] and Gaspar and Santos-Victor [7].

Currently, state-of-the-art computer vision techniques involved in robotic applications such as scene/object recognition, human detection, etc., are computational intensive even with modern hardware. The addition of the unwrapping

Published online: 28 December 2013

process, as such, further compromises the overall processable frame rate of the system. To the best of our knowledge, although various unwrapping techniques have been maturely developed in the field, the issue on real-time processing of the unwrapping process is rarely discussed in the literature. One related work on this matter is an implementation on field-programmable gate array [4]. Their approach had considered a constant look-up table to address the coordinate mapping that is essential in the unwrapping process.

In this work, we demonstrate that the unwrapping process can be efficiently realised to deliver a real-time performance on NVIDIA's graphical processing units (GPUs) with Compute Unified Device Architecture (CUDA). Contrary to [4], a dynamic coordinate mapping was incorporated into our design consideration. The unwrapping routine is first analysed to devise a general approach of parallelisation without considering platform-specific adaptation. However, it was found that the fastest Intel CPU failed marginally to meet the real-time requirements. Therefore, we subsequently introduce optimisation strategies targeted at CUDA platform to take advantage of the hardware capability in general. Finally, the processing speed of the approach is demonstrated on two NVIDIA GPUs with compute capability of 3.0 and 1.1.

In Sect. 2, an unwrapping method proposed for spherical omnidirectional view sensors is briefly explained. Further on, the general parallelisation approach and the CUDA-specific tuning are documented in Sects. 3 and 4, respectively. Results and discussion are provided in Sect. 5, while Sect. 6 concludes this paper.

2 Proposed unwrapping method

An unwrapping technique tailored for spherical COVS was developed by solving the geometry of the ray traces captured by the sensor. It can be conveniently grouped into three key stages.

Initially, the radius of the sphere, R, and the distance measured from its centre to the projection centre, h, are derived using a standard camera calibration procedure [22]. Then, a relationship that maps a world point, $P_{\mathrm{w}}(\rho_{\mathrm{w}}, z_{\mathrm{w}})$, to its corresponding mirror point, $P_{\mathrm{m}}(\rho_{\mathrm{m}}, z_{\mathrm{m}})$, and subsequently image point, $P_{\mathrm{i}}(\rho_{\mathrm{i}}, z_{\mathrm{i}})$, can be observed with Eqs. (1) and (2), respectively. P_{m} are points residing on the surface of a virtual sphere that is used to substitute the actual mirror in the perspective projection model [8]. All the corresponding ρ in the coordinates are the result of a dimension reduction from $P(x, y, z)$ to $P(\rho, z)$ with $\rho = \sqrt{(x^2 + y^2)}$.

$$\sum_{n=0}^{6} a_n \rho_{\mathrm{m}}^n = 0 \tag{1}$$

where the coefficients, a_n, of the polynomial in Eq. (1) are given by

$$\begin{aligned}
a_6 &= 16h^4\left(z_{\mathrm{w}}^2 + \rho_{\mathrm{w}}^2\right)\\
a_5 &= -16h^4R^2\rho_{\mathrm{w}}\\
a_4 &= 4h^2R^2\left(h^2R^2 + 2hR^2z_{\mathrm{w}} + \left(-8h^2 + 2R^2\right)z_{\mathrm{w}}^2\right.\\
&\quad \left.+\left(-8h^2 + 2R^2\right)\rho_{\mathrm{w}}^2\right)\\
a_3 &= 4h^2R^4\left(6h^2 - R^2 - 2hz_{\mathrm{w}}\right)\rho_{\mathrm{w}}\\
a_2 &= R^4\left(-4h^4R^2 + h^2R^4 + \left(-8h^3R^2 + 2hR^4\right)z_{\mathrm{w}}\right.\\
&\quad \left.+\left(-4h^2 + R^2\right)^2 z_{\mathrm{w}}^2 + \left(20h^4 - 12h^2R^2 + R^4\right)\rho_{\mathrm{w}}^2\right)\\
a_1 &= -2hR^6\left(4h^2 - R^2\right)(h - z_{\mathrm{w}})\rho_{\mathrm{w}}\\
a_0 &= -R^6\left(4h^4 - 5h^2R^2 + R^4\right)\rho_{\mathrm{w}}^2
\end{aligned}$$

$$\rho_{\mathrm{i}} = \frac{f}{h - z_{\mathrm{m}}}\rho_{\mathrm{m}} \tag{2}$$

Since the mapping equations are in a closed form, the reverse projection from P_{i} to P_{m} to P_{w} is also possible using the following equations:

$$\rho_{\mathrm{m}} = \frac{fh - \sqrt{-h^2\rho_{\mathrm{i}}^2 + f^2R^2 + \rho_{\mathrm{i}}^2R^2}}{f^2 + \rho_{\mathrm{i}}^2}\rho_{\mathrm{i}} \tag{3}$$

$$z_{\mathrm{w}} = z_{\mathrm{m}} - (\rho_{\mathrm{m}} - \rho_{\mathrm{w}})\left[\frac{hR^4 + \left(R^4 + 2h^2\left(2\rho_{\mathrm{m}}^2 - R^2\right)\right)z_{\mathrm{m}}}{\rho_{\mathrm{m}}\left(R^4 - 4h^2z_{\mathrm{m}}^2\right)}\right] \tag{4}$$

Finally, a projection plane is required to form the final output (i.e. the "unwrapped" image). In this paper, we considered the commonly used yet most computationally expensive cuboid panoramic unwrapping. It uses a projection configuration shown in Fig. 1. This form of unwrapping is comparatively expensive in computation because the $\rho - z$ plane coordinate mapping is unique to every different ρ_{w} in each row of the final output. For cylinder panoramic unwrapping, p_{w} is always kept constant. For the ground plane unwrapping, ρ_{w} is also unique but a ground plane unwrapping generally produces an output that has a much smaller size as compared to the cuboid unwrapping.

The cuboid unwrapping requires three pieces of information to form four identical projection planes—the perpendicular distance of each plane to the optical axis, $y_{\mathrm{w,s}}$, the vertical start location of each plane $z_{\mathrm{w,s}}$, and the corresponding end location, $z_{\mathrm{w,e}}$. Both $z_{\mathrm{w,s}}$ and $z_{\mathrm{w,e}}$ can be deduced from Eqs. (3) and (4) by obtaining two input points, $\rho_{\mathrm{i,s}}$ and $\rho_{\mathrm{i,e}}$. The vertical length of the projection planes is $\lfloor z_{\mathrm{w,e}} - z_{\mathrm{w,s}} \rfloor$ while the horizontal length can be deduced geometrically as $\lfloor 2y_{\mathrm{w,s}} \rfloor$.

Once the projection planes are set up, they are divided into equally sized pixels. The mapping of each pixel of the

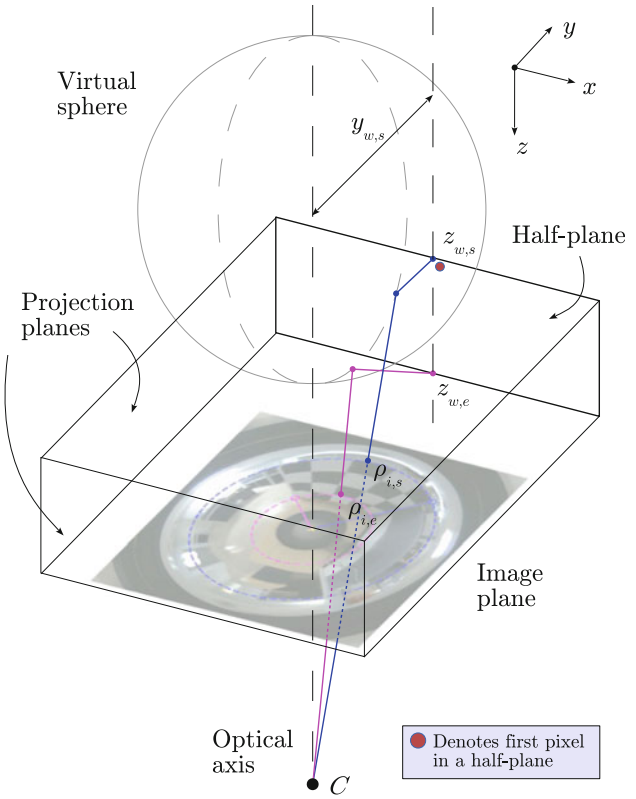

Fig. 1 The cuboid projection planes can be deduced given that the perpendicular distances of each plane to the optical axis, $y_{w,s}$, the vertical start location of each plane, $z_{w,s}$, and the corresponding end location, $z_{w,e}$, are available. Each projection plane consists of two half-planes that are mirror to each other. The *red dot* denotes the first pixel location in a half-plane. C is the centre of projection

projection planes to the image plane is then calculated from Eqs. (1) and (2) using their coordinates in the 3-dimensional space (prior to this, their x and y coordinates have been simplified into ρ). Two look-up tables, LUT_x and LUT_y, that have similar sizes to the projection planes are then used to hold the mapped results.

Given the look-up tables, an unwrapped image as shown in Fig. 2 can then be produced using the common 2-dimensional (2D) data interpolation techniques (e.g. bilinear interpolation). Given a video stream, the unwrapping can proceed on using the same look-up tables until a change in projection planes is needed. For example, changing $y_{w,s}$ will simulate a zooming effect while changing $z_{w,s}$ and $z_{w,e}$ results in a change of the viewing window. The mapping calculation does not necessary be a one-off process. Considering a transition effect (e.g. zooming from near to far), this would require a recalculation of the mapping for every video frame.

3 Parallelisation approach in general

In this section, we describe the parallelisation approach for the said unwrapping method in general. There are basically two processes involved in the unwrapping process—the LUT_x and LUT_y calculations that are carried out on demand, and the data interpolation from the omnidirectional view video stream that is carried out for every frame. However, the mapping procedure is not executed until a change in viewpoint is needed. Therefore, there are two states to be considered. At the "stable state", the system executes the unwrapping using the same LUT_x and LUT_y for each incoming video frame. At the "transition state", the system emulates a smooth transition from one viewpoint to another and therefore the LUT_x and LUT_y have to be recalculated for every frame prior to the data interpolation procedure.

The video frame configuration used in this work has a 24-bit wide pixel size (i.e. 3 colour channels) that uses the common red, green, blue colour model.

3.1 Mapping procedure

The mapping procedure initiates with the calculation of the respective ρ_w that is constant in each column. This calculation is expensive because it involves a square root operation [Complexity of $O(M(n))$] using the Newton Raphson's method (NR) [3]. Subsequently, the mapping procedure proceeds to solve Eq. (1), which is the bottleneck of the process. However, there are several tweaks that enable a high-speed calculation. First, the coefficients of Eq. (1) are rearranged to allow as much constant grouping of R and h as possible. This will enable an off-line compile-time calculation of the relevant constants. In particular, a_4 and a_2 will benefit from this rearranging.

At the end of the entire mapping procedure, it is required to revert ρ_i into x_i and y_i. Therefore, to avoid a second call to the square root operation, it is also useful to calculate the sine and the cosine angle, θ, of each ρ_w at this stage.

$$\sin\theta = \frac{x_\text{w}}{\rho_\text{w}} \quad \cos\theta = \frac{y_\text{w,s}}{\rho_\text{w}} \tag{5}$$

Second, because the pixels in all four of the planes have the exact same coordinates in the 2D $\rho - z$ plane and each half of the projection plane is also a mirror of the other half, the mapping computation can be quantitatively reduced to one-eighth of the original load. This property of the projection planes will be further discussed in Sect. 3.2. In the subsequent part of this paper, this half-portion projection plane is referred to as the half-plane for brevity.

Third, for a qualitative reduction, only one of the roots (i.e. mirror points, ρ_m) of Eq. (1) is valid for the coordinate mapping, while the other five are not useful in this application. Given that there is a good initial guess of the correct root in the first place, a suitable numerical root-finding algorithm can be used to solve the desired root. In fact, since ρ_i, s is known initially, a corresponding ρ_m, s obtained using Eq. (3)

Fig. 2 Using the unwrapping technique [5], a cuboid panoramic unwrapping can be produced

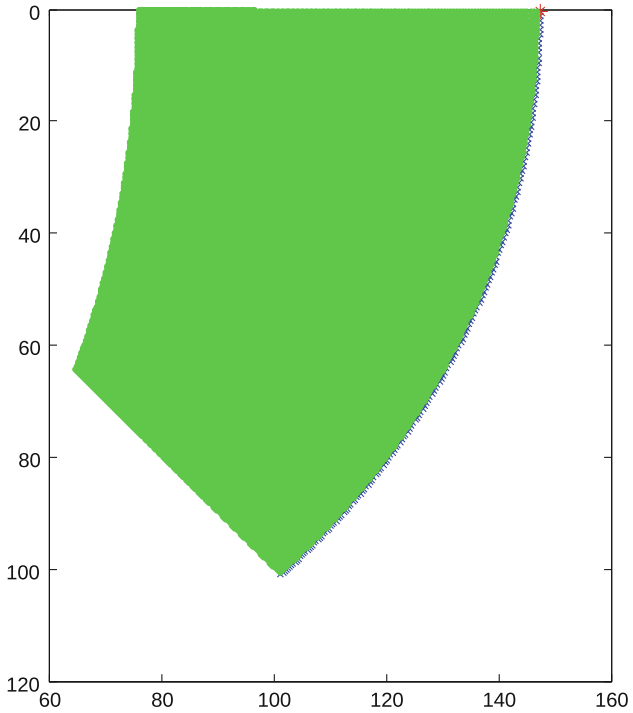

Fig. 3 The mirror points corresponding to a half-plane are densely packed to one another. The *red mark* corresponds to the first pixel in the half-plane, the *blue marks* correspond to the first row, while the *green marks* correspond to the rest of the pixels

serves as a good initial guess for the first pixel's ρ_{m} in the half-plane. As shown in Fig. 3, all the mirror points corresponding to the half-plane's pixels are densely packed to one another. Therefore, if the first pixel is solved, it can then serve as a subsequent initial guess to its neighbouring half-plane pixels.

By exploiting this property, a systematic procedure as shown in Fig. 4 can be devised to solve all the ρ_{m} in the half-plane. As the first pixel's corresponding ρ_{m} is obtained, the pixels in the first row can be solved sequentially. Then, each column in the half-plane can be solved in a parallel manner using the first row as the initial guesses. For a general sense of parallelisation, one can delegate each column to a separate processing thread. Note that theoretically, there is little difference if the propagating direction first starts from the column and then the row. With all the ρ_{m} solved, the corresponding ρ_{i} can be obtained via Eq. (2). Due to the concern with processing speed, NR has been chosen for the relevant procedure since it is known to be the fastest algorithm at the time of writing.

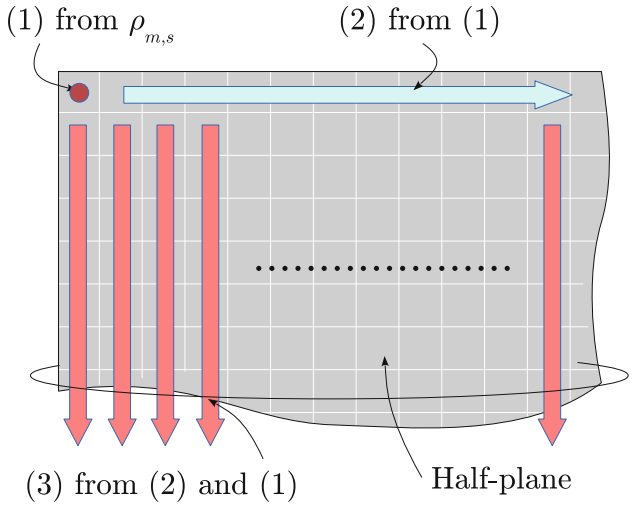

Fig. 4 A systematic procedure is used to solve all the ρ_{m} in the half-plane using ρ_{m}, s. The first pixel (*red dot*) is solved from ρ_{m}, s, then the first row (*blue arrow*) is solved sequentially using the first pixel's ρ_{m}. Finally, each column (*red arrows*) is solved sequentially using the first row's ρ_{m} as the initial guesses

3.2 Data interpolation procedure

The data interpolation procedure is the second step in the unwrapping process. As previously discussed in the mapping procedure, the LUT_x and LUT_y computed are only one-eighth of the required size. Before proceeding to the interpolation process itself, the rest of the mapping values have to be recovered.

Figure 5 illustrates the top view of the complete cuboid projection planes. Let π_n be the nth half-plane in the cuboid where π_1 is the half-plane referred in the mapping procedure, and $(x_{\mathrm{i},1}, y_{\mathrm{i},1})$ be an image point calculated for π_1; the corresponding image points in other half-planes can be resolved as shown in Fig. 5, involving mostly inversion and swapping of the coordinates. With the coordinates of each half-planes resolved, the data interpolation procedure is fairly straightforward since the calculations for each pixel are isolated in this process. The bilinear interpolation method is opted in this procedure.

Parallelisation in this procedure is fairly straightforward. Each processing thread should work on one pixel on the half-plane at a time and completes eight interpolations on the output image, respectively. Apart from that, a standard optimisation can also be applied to the bilinear interpolation function in this procedure where type casting is used in

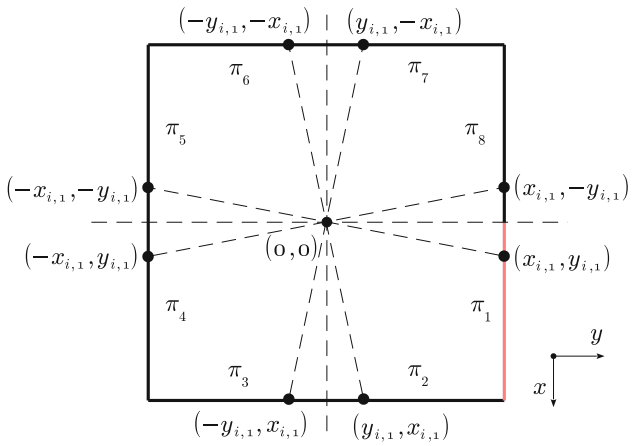

Fig. 5 Top view of the cuboid projection planes with π_n as the nth half-plane. π_1 is the half-plane calculated in the mapping procedure. The corresponding transformations of an image point $(x_{i,1}, y_{i,1})$ in π_1 to the rest of the half-planes are depicted above

favour of the floor and ceiling function (`floor()` and `ceil()`) in the standard C++ mathematics function library. This is possible because the values in LUT_x and LUT_y are all positive and type casting always rounds floating point numbers to the nearest zero. `ceil()` is benefited from type casting because it is equivalent to `floor()` +1.

4 Adaptation on CUDA platform

In Sect. 5, it is shown that, although with the various optimisations introduced, the fastest Intel CPU had marginally failed at meeting a specific real-time requirements. In order to combat with the situation, an alternative using GPU computing was investigated.

In recent years, GPU computing has shown widespread adaptation since the introduction of the NVIDIA GPUs with CUDA capability. ATI (currently AMD) later also responded with STREAM-enabled GPUs that have a similar functionality. Nevertheless, due to a wider availability and technical support community, we have chosen the CUDA platform over STREAM for this adaptation.

The advantages of such GPUs are the massive thread deployment handled by the multi-processors (MP). A typical CUDA application usually allows parallel-running threads in the magnitude of hundreds in total, according to the hardware's capability. Apart from that, GPU computing also benefits image processing problem in general as there are access to hardware-implemented functionalities such as the bilinear interpolation.

However, contrary to common beliefs, the advantage of parallel processing using CUDA platform was not gained merely due to the sheer hardware specification. Instead, careful design that compensates for the limitation of the platform is the major contributor that enables the efficient usage of the MP in CUDA GPUs. In particular, CUDA is prone to issues relating to memory access where a badly designed scheme will usually fail due to non-coalesced accesses. Other than that, the load of each thread should be light in processing footprint due to the limited resource shared by each other.

4.1 Criteria for efficient implementation

An efficient implementation on CUDA platform requires attention on the hardware resource micromanagement as it has the bottleneck often occurring on the data access latency. First, the data reside in the system-wide random access memory (RAM), and the data in the GPU's video RAM (VRAM) are not directly accessible by each other without a high latency memory transfer. Therefore, input data have to be transferred from RAM to VRAM prior to any processing on the GPU kernel. After the data are processed, a reverse transfer is again necessary. In the CUDA's context, the VRAM are termed global memory (GM). In order to avoid further latency, intermediate data transfer of such should be avoided even if one of the stages were performed faster on a CPU. This usually happens when one of the stages in a calculation cannot be effectively designed to run in parallel.

Second, access of GM by the kernel should be kept coalesced for optimum speed. For a GPU with a compute capability of 1.1, each running thread within half of a warp must sequentially access data in 4-, 8- or 16-byte word length and all 16 words should lie in a 64-, 128- or 256-byte segment, respectively [16] (for a compute capability of 3.0, sequential access is not required). In CUDA's context, warps are groups of threads that are simultaneously executed by the MP. If the requirements for coalesced access are not met, each half wrap must issue sixteen 32-byte memory requests for each thread at worse instead of a single coalesced access, thus rendering a penalty of 16× slower speed.

Third, the utilisation of different memory types requires planning. There are basically four memory types in CUDA, each with different latencies, lifetime and advantages associated. The registers are the fastest piece of memory but they are scarce. In addition, they have the lifetime and access scope of a thread, and their utilisation is controlled by a compiler instead of by design. The shared memory (SM) comes in second and has the lifetime and access scope of a block. It is physically divided into 16 banks for a device with a compute capability of 1.1 (32 banks for a compute capability of 3.0) and multiple access to a bank should be avoided to prevent serialised requests. The constant memory comes in third. It is a read-only globally accessible cached memory but with a very limited size. The texture memory

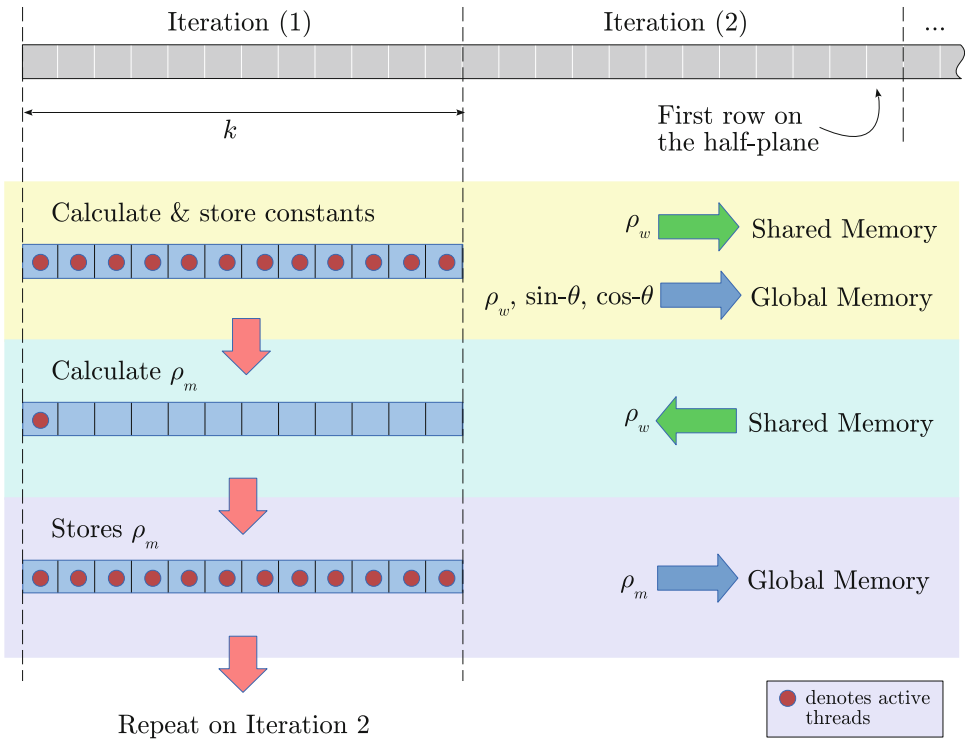

Fig. 6 The sequence design of the GPU kernel for stages 1 and 2 in the mapping procedure. k is the warp size

(TM) and GM have the highest latency. The GM is faster than the TM when access is coalesced. The TM on the other hand is a form of read-only, cached GM that is ideal when coalesced access to the GM is not possible. It also provides hardware-implemented data interpolation if necessary.

4.2 Mapping procedure

The proposed mapping procedure has three stages of calculation. The first stage involves the column-wise constants' preparation, while the second stage calculates the first row's ρ_{m}. Naïvely, the first stage's kernel can be designed so that each thread processes the constants of each column independently. Subsequently, in the second stage, another kernel that can only have one active thread will serially compute the ρ_{m} on the first row. Although this design works, it is not optimum because the first kernel calculates and stores data that are needed by the second kernel to the GM. Thus, it forces an unnecessary fetching of data that will only be used once from the GM during the second kernel's execution.

A more optimised design for stage one and two is to combine them into a single execution. Let k be the warp size of the GPU, this design will require the kernel to have a single block of size $1 \times k$ threads. In the first step of the kernel, each thread calculates the first k column-wise constants, stores them to the GM, and also keeps a copy of the k p_{w} constants in the SM. In the second step, only the first thread will be active and the corresponding k ρ_{m} are calculated from the constants stored in the SM. In the third step, all threads are again active and store the k ρ_{m} to the GM. Finally, the process is repeated for the subsequent k columns until all columns are processed. Figure 6 and Algorithm 1 summarise these two stages.

In the third stage of the mapping procedure, a kernel is designed to have blocks with a single row but columns in the multiple of k. Each thread in each block will be responsible for one column in the half-plane. The execution of this kernel is fairly straightforward where each thread iteratively deduces all the mirror points for their respective columns and, subsequently, their image points. Note that for this entire scheme to work efficiently under coalesced access, unlike the general implementation, the order of propagating direction cannot be arbitrary but instead it must begin in the row's direction and subsequently in the column's. Figure 7 and Algorithm 2 summarise this stage in the mapping procedure.

4.3 Data interpolation procedure

The CUDA version of the data interpolation procedure has a similar design to the general approach described in Sect. 3.2. However, there are three concerns to be addressed. The first concern is related to the memory micromanagement. On the input omnidirectional view image, coalesced access is impossible since bilinear interpolation requires a random access, thus it is best placed in the TM. Another advantage of the TM in this case is the theoretically faster bilinear

Algorithm 1: GPU stage 1 and 2 mapping kernel

```
§ Input: R, h, α, y_{m,s}, width
  Output: ρ_{m,first}
// Get thread position
col ← thread position in x-axis;
// Get number of chunk required
num-chunk ← width/warp-size + (mod(width/warp-size)?1 : 0);
init-guess ← y_{m,s}; // initial guess initialisation
for c ← 0 to num-chunk − 1 do
    x_w ← col + warp-size ∗ c;
    if x_w < width then                          // if position within range
        // global memory constants
        ρ_w ← sqrt(x_w^2 + y_{w,s}^2);
        sin-θ ← x_w/ρ_w;
        cos-θ ← y_{w,s}/ρ_w;
        // shared memory constants
        sm-ρ_w(col) ← ρ_w;
        if col is 1 then                         // use first thread only
            for i ← 1 to warp-size do
                // perpare mirror points in chunk
                sm-ρ_{m,first}(i) ← root-search(R, h, sm-ρ_w(i), z_{w,s}, init-guess);
                init-guess ← sm-ρ_{m,first}(i);
            end
        end
        ρ_{m,first}(x_w) ← sm-ρ_{m,first}(col);
    end
end
```

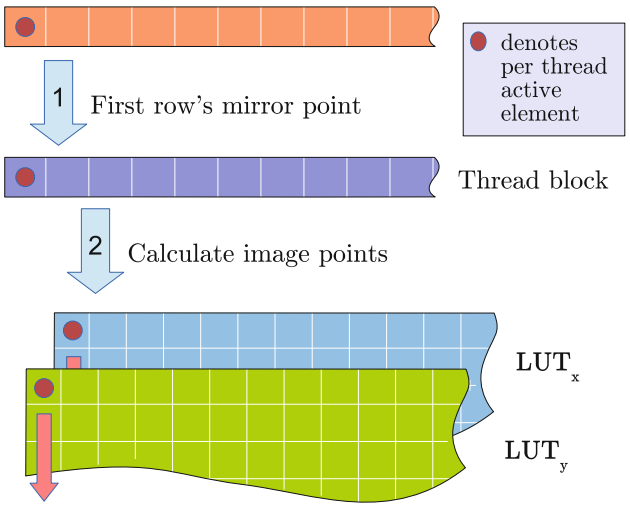

Fig. 7 The sequence design of the GPU kernel of stage three in the mapping procedure

interpolation that can be performed by the hardware function. Various works (e.g. [2, 13, 18, 20, 21]) had also reported a similar efficiency of the in-built bilinear interpolation. On the output image, it has to be allocated in the GM since it has abundant space and is writable.

The second concern is related to the optimisation for coalesced access of the unwrapped image stored on the GM. In general, the coalesced access is achieved if the unwrapped image is arranged so that each of the 32 consecutive pixels (assuming 4 colour channels) in the image is aligned on a 128-byte block. While this arrangement is naturally obtained for most cases of applications, a special preconditioning is required to achieve it in this data interpolation procedure.

The reason for this is due to the partitioning of the projection planes in the mapping procedure. Consider a projection plane with π_8 and π_1 as an example, where π_8 is the mirror of π_1. The unwrapped image referred π_1 from the look-up table. However, because the width of the projection plane is not constrained to a multiple of 32 pixels, the access to the unwrapped image space is always offset by $\mathrm{mod}\,(w/32)$ where w is the width of the projection plane. On other projection planes, the offsets are accumulated due to the same reason.

In order to correct the alignment, a preconditioning has to be applied to properly insert paddings between the projection planes as shown in Fig. 8. These paddings will eventually become column gaps in the unwrapped image. However, because there is no perspective continuity among the projection planes, the gaps do not raise any additional issue other than that of appearance-wise. If the gaps must be removed for some reasons, it is still achievable without performance penalty. When the image is to be transferred from the global memory to the RAM, the process is separated into four iterations, with each carrying one-forth portion of the unwrapped image offset by the paddings and thus indirectly removes the gaps.

Algorithm 2: GPU stage 3 mapping kernel

```
Input: R, h, ρ_w, ρ_{m,first}, z_{w,s}, width, height, warp-size
Output: LUT-x, LUT-y
// Get thread position
x_w ← thread position in x-axis;
init-guess ← ρ_{m,first}(x_w); // initial guess initialisation
if x_w < width then                              // if position within range
    // get first row's image points
    LUT-x(x_w, 1) ← mirror2image(init-guess) × sin-θ(x_w);
    LUT-y(x_w, 1) ← mirror2image(init-guess) × cos-θ(x_w);
    for j ← 2 to height do
        // get other rows' mirror points
        ρ_m ← root-search(R, h, ρ_w(x_w), z_{w,s} + j, init-guess);
        init-guess ← ρ_m;
        // get other rows' image points
        LUT-x(i, j) ← mirror2image(ρ_m) × sin-θ(x_w);
        LUT-y(i, j) ← mirror2image(ρ_m) × cos-θ(x_w);
    end
end
```

The third concern is related to the number of colour channel in the unwrapped image. Previously in the second concern, the example assumes a 32-bit image that has four colour channels. The 32-bit image coalesced naturally

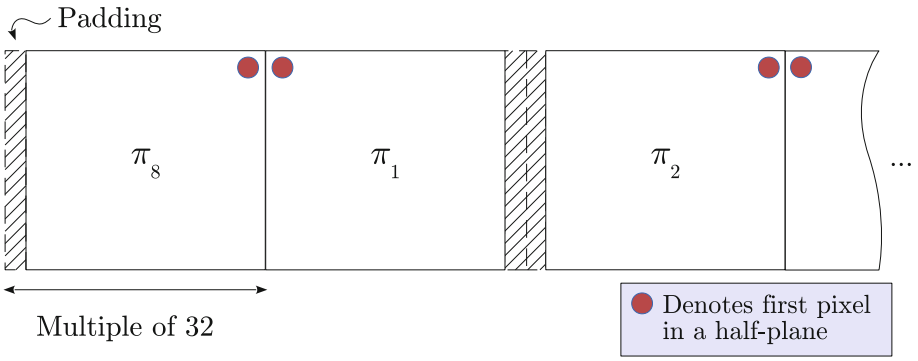

Fig. 8 Paddings are added to the unwrapped image so that coalesced access to the global memory is achieved

when the above preconditioning is applied. However, for a 24-bit image to comply with the requirement, the output for a 24-bit image needs to be converted into a 32-bit image (i.e. by adding a dummy channel). The extra colour channel is usually used to represent transparency in the image and can be safety set to full opacity. Since the GM has space redundancy, the extra colour channel is justifiable for the speed gained.

On the kernel implementation, each thread processes a single pixel in each half-plane in the output image and basically follows the design of the general approach. First, the input image is split into multiple 2D arrays of its own colour channel and loaded to the TM. Then, each thread resolves the required coordinates from LUT_x and LUT_y, and subsequently interpolates data from the TM. After each colour channel is processed, they are stored together as a 4-byte `uchar4` data type [16] to the allocated GM to achieve coalesced access. Again, for the purpose of coalescing, the kernel blocks are designed with widths in the multiple of the warp size. However, note that due to the mirroring effect in each subsequent half-plane as discussed in Sect. 3.2, GPUs with compute capability of 1.1 can only maintain coalescing at 50 % of the time (refer Sect. 4.1). A summary of the GPU kernel implementation is provided in Fig. 9 and Algorithm 3.

Algorithm 3: GPU data interpolation kernel

Input: input-red, input-green, input-blue, `LUT-x`, `LUT-y`, width, height, c_x, c_y
Output: `unwrap-img`

```
1  j ← thread position in x-axis;
2  i ← thread position in y-axis;
3  if i < width and j < height then
4      for k ← 0 to 3 do                                  // odd-numbered half-planes
5          x ← plane-transf(π(1+2k),LUT-x(i,j));
6          y ← plane-transf(π(1+2k),LUT-y(i,j));
7          px-red ← tex2D(input-red,x,y);
8          px-blue ← tex2D(input-blue,x,y);
9          px-green ← tex2D(input-green,x,y);
10         px-alpha ← 255;                  // full opacity on alpha channel
11         unwrap-img(i+offset(π(1+2k)),j) ←
           uchar4(px-red,px-green,px-blue,px-alpha);
12     end
13     for k ← 0 to 3 do                                  // even-numbered half-planes
14         x ← plane-transf(π(2+2k),LUT-x(i,j));
15         y ← plane-transf(π(2+2k),LUT-y(i,j));
16         px-red ← tex2D(input-red,x,y);
17         px-blue ← tex2D(input-blue,x,y);
18         px-green ← tex2D(input-green,x,y);
19         px-alpha ← 255;                  // full opacity on alpha channel
20         unwrap-img(i+offset(π(2+2k)),j) ←
           uchar4(px-red,px-green,px-blue,px-alpha);
21     end
22 end
```

5 Results and discussion

5.1 Requirements for real-time performance

The requirements set to qualify the unwrapping process as real-time are that (1) a reasonable frame rate is reached and (2) the data processed in each frame exceed a resolution that is considered reasonably high to date. As of the writing of this paper, most Universal Serial Port powered cameras (commonly known as "webcams") have an average frame-per-second (fps) performance of 25. For the second requirement, the high-definition (HD) 720p standard of 1,280 × 720 pixels on each projection plane is considered.

The mapping procedure was tested on maps with sizes 180 × 180, 360 × 360, and 720 × 720 pixels. Similarly, the data interpolation procedure was tested on output images with sizes 1,440 × 180, 2,880 × 360, and 5,760 × 720 pixels. Note that the largest image size used in the data interpolation procedure exceeded the HD 720p resolution in all projection planes. All results in the experiments were obtained by averaging over 100 iterations to reduce the effect of unaccountable factors during real-time execution.

5.2 Hardware specification

The general parallelisation discussed in Sect. 3 will be tested using Intel Core i7-3770 that has a stock clock speed of 3.4 GHz, which is by far the fastest one from Intel. It has

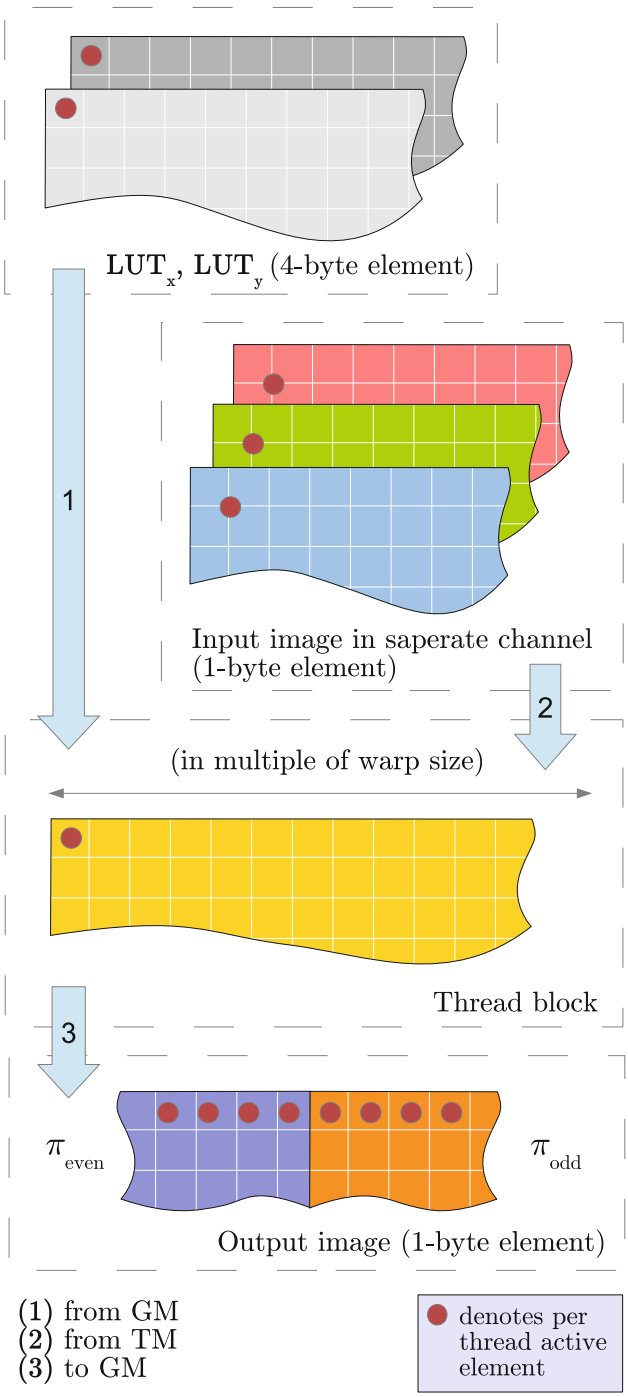

Fig. 9 Implementation of the data interpolation procedure in CUDA

four physical cores but support eight concurrent threads due to Intel's "Hyper Threading" technology. Multi-core processing on Intel CPU is enabled via the use of the "Threading Building Block" library [10].

The CUDA adaptation in Sect. 4 on the other will be tested on a high-end and a low-end NVIDIA GPUs. NVIDIA GeForce GTX 645 is a high-end GPU that has three MPs with compute capabilities of 3.0, while NVIDIA GeForce 9500 GS is a low-end GPU with four MPs of compute capability 1.1. The compute capability mainly translates to the arithmetic throughput of each MP with 192 and 8 operations per clock cycle [16], respectively on the GTX 645 and the 9500 GS. The total MP throughput, also commonly known as the total "CUDA cores", therefore is 576 and 32, respectively on each device. The clock rates for both GPUs are 834 and 1,375 MHz, respectively.

Table 1 Specifications of the GPUs in test

	GeForce GTX 645	GeForce 9500 GS
Clock rate (MHz)	834	1,375
Compute capability	3.0	1.1
PCI-express	3.0	1.1
CUDA core	576	32
Multi processor	3	4
Warp size	32 threads	
Global memory (MB)	1,048	512
Shared memory per MP (KB)	48	16
Register (32 bit) per block	65,536	8,192
Constant memory (B)	65,536	

Apart from that, it is also noteworthy that the 9500 GS uses the more inferior PCI-Express 1.1 data bus that will limit the memory transfer speed and this will be reflected in the less optimised performance in the memory transfer between the GM and the RAM. Nevertheless, although the 9500 GS is significantly lacking in terms of hardware capability, it has the advantage of been more cost effective at the time of writing and thus is included in the comparison. Table 1 summarises the various noteworthy specifications of the GPUs in test.

5.3 Kernel block size optimisation

Several GPU kernels discussed in Sect. 4 have no restriction on one or more of their block dimensions. In general, apart from factors due to warp size, design restriction, and hardware limitation, the selection of a kernel block's dimension should also take the GPU's MP occupancy into account. Since each MP can process multiple number of warps simultaneously, the dimensions of the block can be adjusted to obtain an optimum configuration that fully occupies the MPs' resources. In theory, this optimisation helps in reducing the effect of memory access latency for a kernel that has a relatively low computational load [16]. Nevertheless, empirical experiments that exhaustively test on different configurations are needed to confirm the actual optimisation gain, if any. The largest data sizes (720 × 720 and 5,760 × 720 pixels, respectively) were used for this part of the experiments to contrast a more significant difference in the performance.

The MP occupancy is basically calculated from the number of warp executable by each MP versus its allowed maximum as listed in Table 1 due to a certain block size configuration. The total executable warp, apart from the block size, is also affected by the registers and SM available to each MP. In general, a larger block allows more concurrent warps but at the same time demands more registers. As more registers are occupied and more threads are designated (i.e. larger block), the number of block allowed in each MP will become limited instead. A more comprehensive visualisation can be obtained using the NVIDIA's CUDA GPU occupancy calculator [15].

The box plots in Fig. 10 show the performances of the second mapping kernel with varying block sizes. There were drops in occupancy for 9500 GS at 92 and 160 threads, and GTX 645 at 160 threads mainly due to a limited hit in register usage. In general, the speed performance of this kernel was consistent with minute fluctuation (0.16 ms fluctuation on average) occurred across different dimension configurations and a slight advantage at 64 and 224 threads for the GeForce 9500 GS. Due to this reason, it is assumable that the fluctuation is statistically trivial and that processing speed was mostly stable in different configurations. The GTX 645 also showed consistent processing speed (7 µs fluctuation on

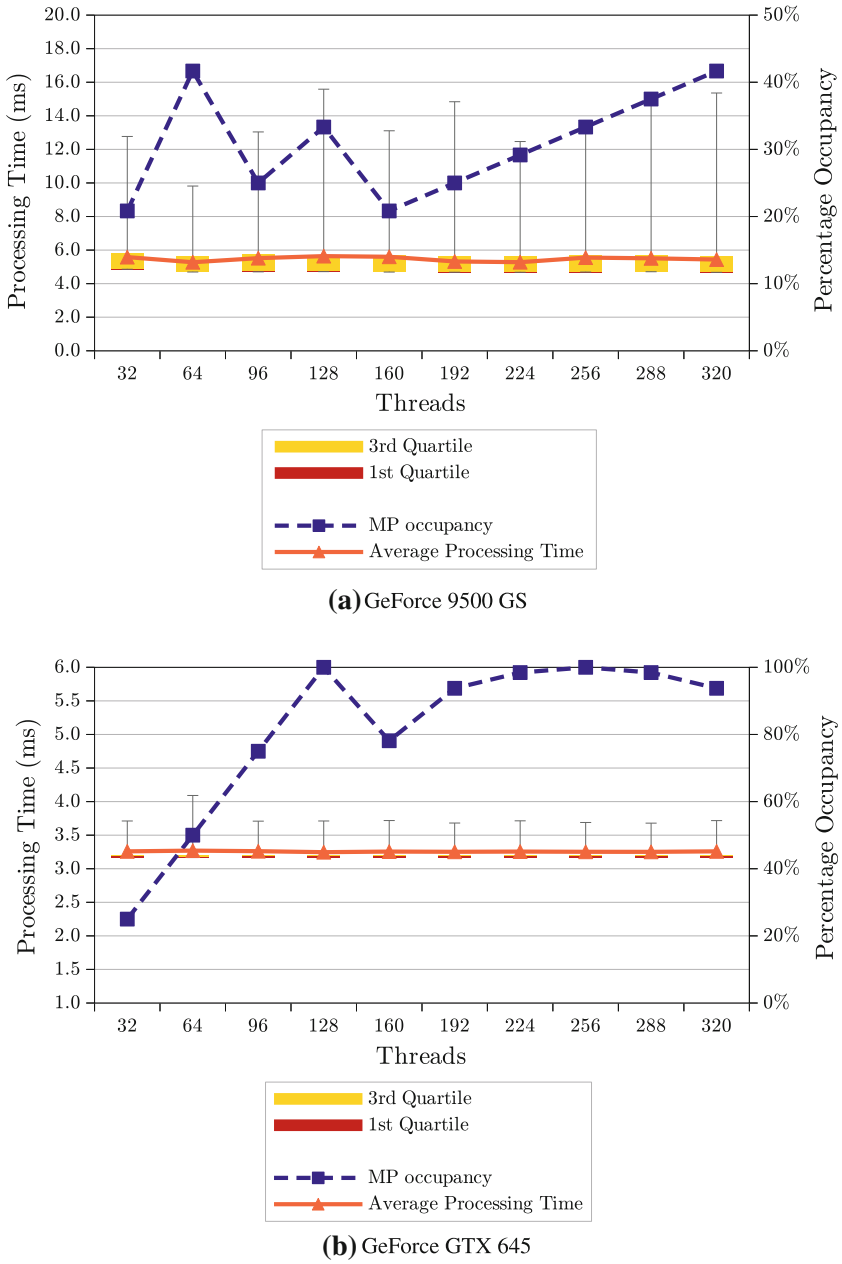

Fig. 10 The second mapping kernel's performances with varying dimensions for both GPUs using a 720 × 720 pixels half-plane

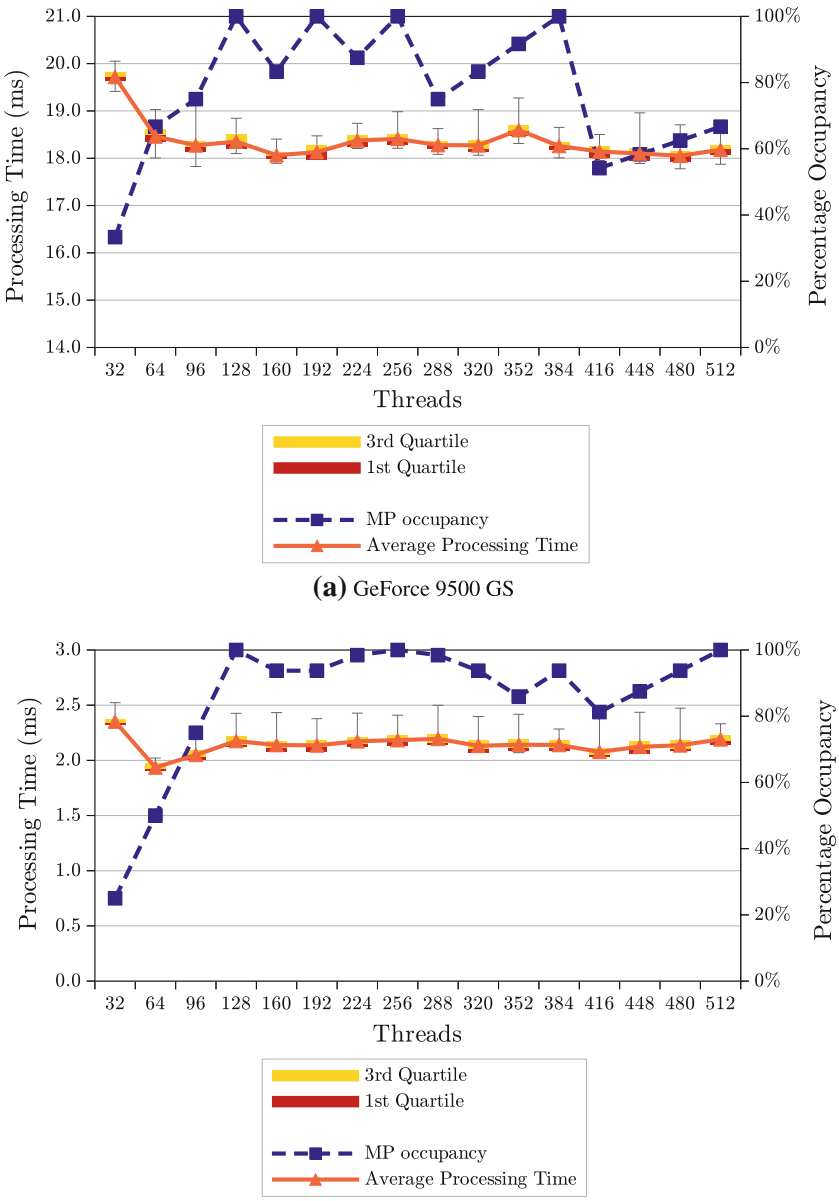

Fig. 11 The data interpolation kernel's performance under varying dimension for both GPUs. The data used were a 24-bit 5,760 × 720 pixels image

average) with no obvious advantage at either thread configuration. These observations were anticipated because the kernel was designed to work in full coalescing,[1] thus keeping the latencies caused by the memory transfer at minimum.

The performances of the data interpolation kernel for both GPUs were somewhat different initially as shown in box plots of Fig. 11. First, on the occupancy, the drops across the different configurations were mainly due to the limitation hit on the maximum allowable threads in a MP. Onto the performance, for the 9500 GS, it began at a less optimum speed at 32 threads but improved at the subsequent thread block sizes. The performance was again observed to be consistent with minor fluctuation afterwards. For the GTX 645, its processing speed was optimum at 64 threads but worsened at larger block size and saturated afterwards. The reason for this could be due to the limitation in image segmentation. Since the output image was processed by segmenting it into a size similar to the kernel block, if the segmentation ended with an odd number segment, one of the processing slots would be idle eventually at run-time. The risk of increasing the block size is that if the mentioned situation occurred, the idling slot would be idle for a longer period because each segment would have more data to be processed. On the other hand, at 32 threads, the main contributing factor would be the MP

[1] The coalesced access of the unwrapping kernel can be confirmed using the NVIDIA visual profiler [17].

occupancy. The case of the 9500 GS was in fact similar to the GTX 645 but appeared differently because its efficiency at 32 threads was far worse than that of the GTX 645, and there was no obvious optimum thread block size at subsequent sizes as they saturated too soon.

On a separate note, the graphs show only the total thread count because it was observed that all possible vertical and horizontal dimension combinations that produced the same number of threads had a similar performance as long as the horizontal dimension was in a multiple of the warp size. In general, the improvement gained from this kernel block optimisation was marginal ($\sim$1 or 2 ms), albeit the best configurations were obtained. This indirectly implies an effective design of the respective kernels.

5.4 Performance of the unwrapping process

The performances of the mapping and data interpolation procedures in the unwrapping process are separately compared in Figs. 12 and 13, respectively. The kernel block size optimisation was also applied in this comparison. Note that during the GPU data interpolation, the gaps in the unwrapped image were eliminated using the memory transfer trick discussed in Sect. 3.

As shown in Fig. 14, the data interpolation procedure running on the GPUs requires a memory transfer prior to and after kernel execution. An omnidirectional image from a video stream needs to be copied over to the GM before it is unwrapped, after which the unwrapped output is transferred back to the RAM so that it is accessible by a CPU. However, at times, applications may require further processing (e.g. computer vision applications) after the image is unwrapped. Thus, the final data transfer can be avoided provided that any further processing is completed in the GPU. If this is the case, there can be an additional saving on the processing time by approximately 50 %.

5.5 Summaries

Tables 2 and 3 summarise the performances of the unwrapping process expressed in fps respectively under the two different states (stable and transition) discussed in Sect. 3.

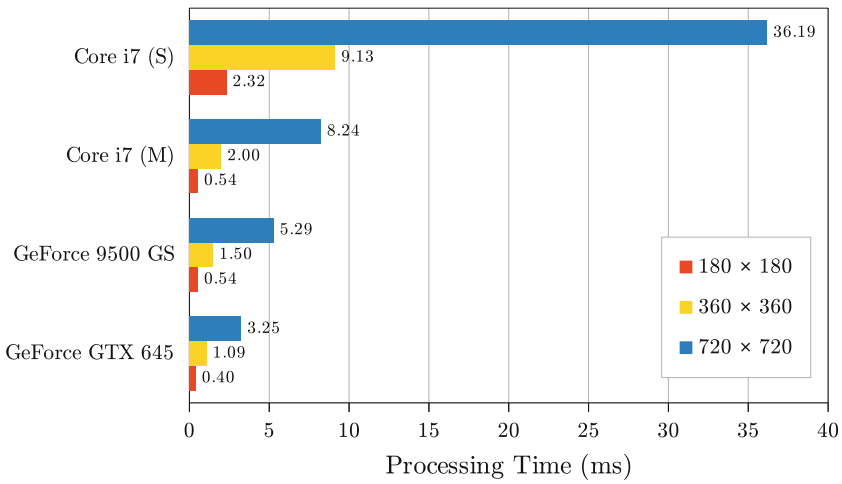

Fig. 12 The performances of the mapping procedure using various hardware. *S* and *M* refer to the single-core and multi-core environments, respectively

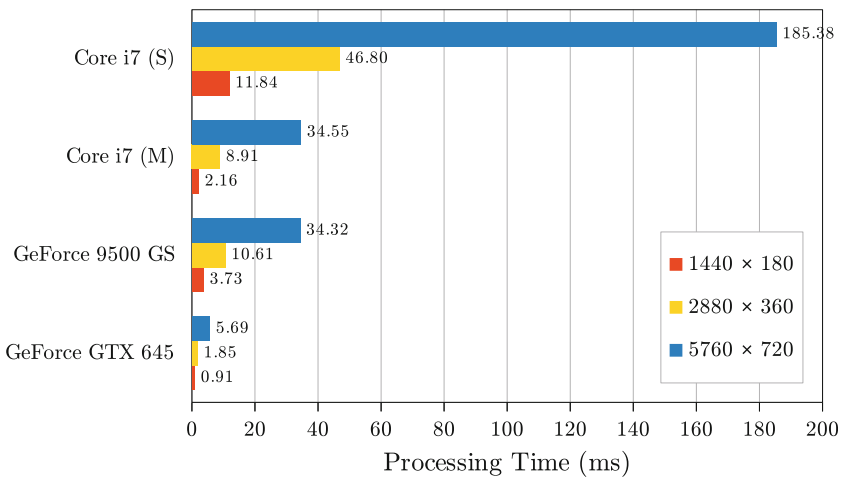

Fig. 13 The performances of the data interpolation procedure using various hardware. *S* and *M* refer to the single-core and multi-core environments, respectively

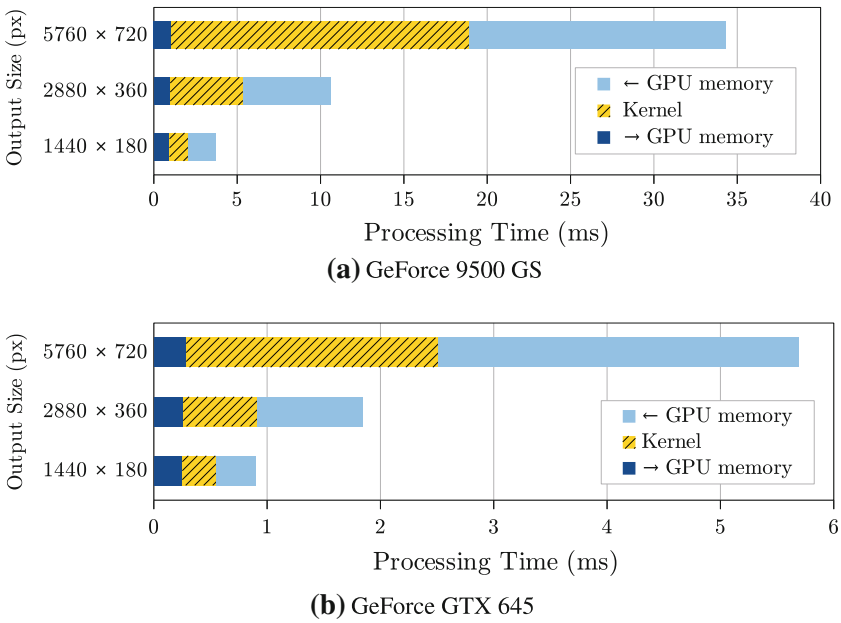

Fig. 14 The breakdown of the GPU data interpolation kernel processing speed for a 24-bit image

Table 2 The performance of the unwrapping process (fps) under stable state

Output size (px)	Performance (fps)		
	1,440 × 180	2,880 × 360	5,760 × 720
Core i7 (M)	462.87	112.19	**28.94**
Core i7 (S)	84.49	21.37	5.39
GTX 645	1,104.84	541.85	**175.61**
9500 GS	267.90	94.22	**29.13**
GTX 645 (T)	1,806.53	1,096.32	**397.92**
9500 GS (T)	482.88	186.94	**52.73**

A boldfaced value denotes real-time performance

S single-core, *M* multi-core, *T* with memory transfer advantage (i.e. no output memory transfer)

Table 3 The performance of the unwrapping process (fps) under transition state

Output size (px)	Performance (fps)		
	1,440 × 180	2,880 × 360	5,760 × 720
Core i7 (M)	370.06	91.65	23.37
Core i7 (S)	84.49	21.37	5.39
GTX 645	765.62	340.77	**111.81**
9500 GS	233.88	82.58	**25.25**
GTX 645 (T)	1,047.60	499.70	**165.41**
9500 GS (T)	382.59	146.08	**41.24**

A boldfaced value denotes real-time performance

S single-core, *M* multi-core, *T* with memory transfer advantage (i.e. no output memory transfer)

Both GPUs were delivering performances meeting the real-time requirements. On the largest image size, the 9500 GS had a marginal real-time performance when it was executed without the memory transfer advantage. This is largely due to the fact that the hardware's technology is about two generations out of date and there is a weakness with memory coalescing as discussed in Sect. 3. On the other hand, the GTX 645 was managed to perform at an over-specification condition. This is important as it will allow the system to execute other computer vision processes after the unwrapping procedure.

On the other hand, the Core i7 was only able to keep up with the real-time performance on the stable state under multi-core environment. In addition, its performance is worse than 9500 GS even when the GPU has a disadvantage on memory transfer. Considering the cost of the CPU versus this two-generation old GPU, the price/performance ratio of the Core i7 in general is not satisfactory for the purpose of this unwrapping problem. Keeping aside this issue, there is a high chance that the CPU is also unable to execute any subsequent computer vision processes without sacrificing its fps performance.

6 Conclusion

In this paper, we have proposed a real-time implementation approach for cuboid panoramic omnidirectional view image unwrapping. Initially, we discussed a general parallelisation of the said process where the NR algorithm is adopted into a parallel deployable pattern. Subsequently, we devised various optimisation to exploit the platform advantage of CUDA devices on the proposed parallelisation scheme.

Experiments showed that the Core i7 barely maintained real-time performance at the stable state while marginally failed during the transition state. On the other hand, the

GTX 645 had considerably surpassed the specification of real-time implementation set, thus allowing further executions of other computer vision algorithms when necessary. At the most tasking transition state, the GTX 645 was able to deliver outputs that exceed the HD 720p resolution at approximately 165 and 112 fps, with and without a memory transfer advantage, respectively. The two-generation old 9500 GS, albeit marginal, was also producing a real-time performance at 25 and 41 fps.

For future work, we will integrate the state-of-the-art visual detection algorithms into existing platforms by taking advantage of the over-specification performance.

Acknowledgments Nguan Soon Chong thanks Swinburne University of Technology (Sarawak Campus) for his Ph.D. studentship. The authors would like to personally thank Jian Soon Ng and Albin Sui Hian Kuek for their useful discussion on this research topic.

References

1. Baker, S., Nayar, S.K.: A theory of single-viewpoint catadioptric image formation. Int. J. Comput. Vis. **35**(2), 175–196 (1999)
2. Bui, P., Brockman, J.: Performance analysis of accelerated image registration using gpgpu. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, GPGPU-2, pp. 38–45 (2009). doi:10.1145/1513895.1513900
3. Burden, R.L., Faires, D.J.: Numerical Analysis, 7th edn. Brooks Cole, Belmont (2000)
4. Chen, L.D., Zhang, M.J., Xiong, Z.H.: Series-parallel pipeline architecture for high-resolution catadioptric panoramic unwrapping. IET Image Process. **4**(5):403–412 (2010). doi:10.1049/iet-ipr.2009.0286
5. Chong, N.S., Kho, Y.H., Wong, M.L.D.: A closed form unwrapping method for a spherical omnidirectional view sensor. EURASIP J. Image Video Process. **2013**, 5 (2013). doi:10.1186/1687-5281-2013-5, url:http://jivp.eurasipjournals.com/content/2013/1/5
6. Chong, N.S., Kho, Y.H., Wong, M.L.D.: Custom aspect ratio correction for unwrapped omnidirectional view images. Comput. Electr. Eng. (2013, in press). doi:10.1016/j.compeleceng.2013.04.005, url:http://dx.doi.org/10.1016/j.compeleceng.2013.04.005
7. Gaspar, J., Santos-Victor, J.: Visual path following with a catadioptric panoramic camera. In: Proceedings of the International Symposium on Intelligent Robotic Systems—SIRS'99, Coimbra (1999). url:http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.3379
8. Hartley, R.I., Zisserman, A.: Multiple View Geometry in Computer Vision, 2nd edn. Cambridge University Press, Cambridge (2004)
9. Hicks, R.A., Bajcsy, R.: Reflective surfaces as computational sensors. Image Vis. Comput. **19**(11), 773–777 (2001). url:http://linkinghub.elsevier.com/retrieve/pii/S0262885600001049
10. Intel Corporation: Intel threading building blocks. Version 4.0 (2012)
11. Jeng, S.W., Tsai, W.H.: Using pano-mapping tables for unwarping of omni-images into panoramic and perspective-view images. Image Process. IET **1**(2), 149–155 (2007). doi:10.1049/iet-ipr:20060201
12. Lei, J., Du, X., Zhu, Y.F., Liu, J.L.: Unwrapping and stereo rectification for omnidirectional images. J. Zhejiang Univ. Sci. A **10**(8), 1125–1139 (2009). url:http://www.springerlink.com/index/10.1631/jzus.A0820357
13. Li, J., Lu, Y., Pu, B., Xie, Y., Qin, J., Pang, W.M., Heng, P.A.: Accelerating active shape model using gpu for facial extraction in video. In: Proceedings of the IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS 2009), vol. 4, pp. 522–526 (2009). doi:10.1109/ICICISYS.2009.5357636
14. Nayar, S.K.: Catadioptric omnidirectional camera. In: Proceedings of the 1997 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 482–488 (1997). doi:10.1109/CVPR.1997.609369
15. NVIDIA Corporation: CUDA GPU occupancy calculator. Version 3.0 (2012a)
16. NVIDIA Corporation: NVIDIA CUDA C Programming Guide. NVIDIA Corporation, Santa Clara (2012b)
17. NVIDIA Corporation: NVIDIA visual profiler. Version 4.2 (2012c)
18. Rossi, R., Savatier, X., Ertaud, J.Y., Mazari, B.: Real-time 3d reconstruction for mobile robot using catadioptric cameras. In: Proceedings of the IEEE International Workshop on Robotic and Sensors Environments (ROSE 2009), pp. 104–109 (2009). doi:10.1109/ROSE.2009.5355981
19. Scaramuzza, D., Martinelli, A., Siegwart, R.: A toolbox for easily calibrating omnidirectional cameras. In: Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, Beijing, pp. 5695–5701 (2006). doi:10.1109/IROS.2006.282372
20. Scherl, H., Keck, B., Kowarschik, M., Hornegger, J.: Fast gpu-based ct reconstruction using the common unified device architecture (cuda). In: Proceedings of the IEEE Nuclear Science Symposium Conference Record (NSS '07), vol. 6, pp. 4464–4466 (2007). doi:10.1109/NSSMIC.2007.4437102
21. Zhang, H., Xie, Y., Heng, P.A.: Accelerating feature extraction for patch-based multi-view stereo algorithm. In: Proceedings of the International Conference on Computer Design and Applications (ICCDA), vol. 5, pp. 511–515 (2010). doi:10.1109/ICCDA.2010.5541068
22. Zhang, Z.: A flexible new technique for camera calibration. IEEE Trans. Pattern Anal. Mach. Intell. **22**(11), 1330–1334 (2000). doi:10.1109/34.888718

Nguan Soon Chong received his BEng(Hons) in Robotics and Mechatronics from Swinburne University of Technology in 2009. He is currently a Ph.D. student in the same institute with support from a Ph.D. studentship. Prior to doing his Ph.D., he worked as a research assistant involving applications of CUDA in watermarking algorithm.

His research interests include omnidirectional view sensors and image processing algorithms.

M. L. Dennis Wong received his BEng(Hons) in Electronics and Communication Engineering from the Department of Electrical Engineering and Electronics, University of Liverpool, Liverpool, UK, in 1999. Then, he joined the Signal Processing and Communication Group in the same department as a Ph.D. student. He received the Ph.D. from the same institution in 2004. In 2004, Dr. Wong joined the Faculty of Engineering, Swinburne University of Technology (Sarawak Campus) (SUTS) as a Lecturer. Subsequently, he was appointed as a Senior Lecturer in 2007 at the same institution. In 2012, Dr. Wong was appointed as an Associate Professor and subsequently in 2013, as the Dean for Faculty of Engineering, Computing and Science.

His research interests include statistical signal processing and pattern classification, machine condition monitoring, and VLSI for digital signal processing.

Yau Hee Kho received his BEng (first class honours) degree in 1997, and the Ph.D. degree in 2008, all from the University of Canterbury, Christchurch, New Zealand. He worked as a R&D/RF Engineer in Singapore, and Research Associate in New Zealand. He was in Swinburne University of Technology (Sarawak Campus) as a Senior Lecturer from 2009 to July 2013. He is now an Assistant Professor with the Nazarbayev University in Astana, Kazakhstan.

His research interests include signal processing, wireless communications and industrial electronics.