Acceleration of Stereo-Matching on Multi-core CPU and GPU

Tian Xu
School of Computing Science
University of Glasgow
t.xu.1@research.gla.ac.uk

Paul Cockshott
School of Computing Science
University of Glasgow
William.Cockshott@glasgow.ac.uk

Susanne Oehler
School of Computing Science
University of Glasgow
Susanne.Oehler@glasgow.ac.uk

***Abstract*—This paper presents an accelerated version of a dense stereo-correspondence algorithm for two different parallelism enabled architectures, multi-core CPU and GPU. The algorithm is part of the vision system developed for a binocular robot-head in the context of the CloPeMa [1] research project. This research project focuses on the conception of a new clothes folding robot with real-time and high resolution requirements for the vision system. The performance analysis shows that the parallelised stereo-matching algorithm has been significantly accelerated, maintaining $12\times$ and $176\times$ speed-up respectively for multi-core CPU and GPU, compared with non-SIMD single-thread CPU. To analyse the origin of the speed-up and gain deeper understanding about the choice of the optimal hardware, the algorithm was broken into key sub-tasks and the performance was tested for four different hardware architectures.**

***Keywords*—GPU; Multi-core CPU; Acceleration; Stereo matching; Robotic vision; Dense-correspondences;**

I. Introduction

Dense-correspondence stereo-matching algorithms are an essential element in computer vision, widely used in image and video processing, as well as for robotic vision. Stereo matching uses two images from two slightly different viewpoints of a scene, here referred to as left and right images. The stereo-matching algorithm then maps each pixel in the left image to the corresponding pixel in the right image, using correlation or sum of squared differences to determine the optimal match. The disparity for each pixel in the left image to its corresponding pixel in the right image are recorded in form of a dense-correspondence map. When the intrinsic and extrinsic camera parameters are known, a depth map or 3D reconstruction of the scene can be computed. Depth data derived from stereo-systems have been used previously for applications in a variety of domain, such as passive navigation, cartography, surveillance. More specifically, with respect to the scenario of robotic cloth manipulation, image stereo matching has been proven to be effective for cloth classification [21] and cloth grasp point detection [11].

When evaluating stereo-matching algorithm for high-resolution images, two aspects are often of interest, accuracy and computational costs. In the literature, improvement of image matching algorithms accuracy is a common subject, for instance, reducing match failures in real-world data with image noise reduction and detection of false minima are widely discussed [14]. Different approaches are in use as how to accurately locate the best corresponding pixel in the right image [16][23], however stereo-matching algorithms remain inherently computationally expensive. With large high resolution images, this becomes an increasing problem, especially in systems with real-time requirements. In the robotics CloPeMa project, an image size of 4928×3264 pixels is used. This means that the matching algorithm needs to cope with millions of pixels in a very short amount of time, in order to ensure adequate response time of the robotic system. This paper is therefore focused only on the improvement of the execution time using parallel programming versus trying to reduce the computational costs. Also the accuracy to which the dense-correspondence algorithm performs is considered as sufficient for the tasks of the binocular robot head used in the project.

With recent advances in parallelism, the performance of CPUs and GPUs has rapidly increased. As a step towards a real-time vision system, we decided to implement a parallel dense stereo-correspondence algorithm. However, different hardware architectures have different performance characteristics and can require different code optimisation to take full advantage of them. Hence, in this paper, two different implementations of the same dense stereo-correspondence algorithm are presented, one implemented in Vector Pascal using a SIMD computation model for multi-core CPU architectures and one in CUDA for Nvidia GPUs.

The aim is to significantly improve the execution time using parallel programming, to bring the response time of robot-vision system closer to real-time. Assessing the performance of the algorithm for two specific parallel environments (i.e. multi-core CPU and GPU), the contributions of the paper are two-fold:

- The demonstration of a successful parallelisation of the dense-correspondence algorithm, achieving acceptable execution times over high resolution images on both CPU and GPU for the binocular robot-head.
- An in-depth analysis of the acceleration impact of the individual key sub-parts of the algorithm in relation to the type of hardware used. This permitted us to highlight the parts with the greatest impact on the overall performance of the algorithm as well as to demonstrate their potential speed-up limits.

In an industrial context, a faster vision system would make clothes manipulation robots more viable.

The paper is organized as follows: We first review various

[1] **Clo**thes **Pe**rception and **Ma**nipulation project: www.clopema.eu

existing related work, i.e. image matching algorithms, as well as existing attempts to parallelize image matching in different system architectures in Section II. Section III reviews our stereo matching algorithm, applied to CPU and GPU parallel architectures. In Section IV, we report preliminary findings for the accelerating image matching algorithms under two parallel system architectures (multi-core CPU and GPU) and discuss the implications of our findings. Finally, we conclude the paper in Section V.

II. Related Work

We start by introducing various matching algorithms in prior art (Sec. II.A). Next, we review various approaches using multi-core CPUs and GPUs to accelerate matching algorithms (Sec. II.B).

A. Image Matching Algorithms

Stereo image matching entails discovering the most likely matches between pixels in two images. Typically these are captured simultaneously from cameras in different spatial locations, but a similar algorithmic problem exists with sequential image capture from aircraft or from orbiting cameras. The technique is widely used in computer vision and robotics, including: data visualization, three dimensional map building and robot pick and place. It has been studied over several decades in computer vision and many researchers have worked at solving it [14].

There are two main classes of stereo matching algorithms: local methods and global methods. Local algorithms are statistical methods and are usually based on correlation. For global algorithms, the task of computing disparities is cast in terms of energy minimization, and is solved by various optimization techniques [9][10]. Compared with local algorithms, global algorithms are normally computationally much more expensive.

Local algorithms can be subdivided into two categories: feature-based algorithms and area-based algorithms. Feature-based matching algorithms [1][19] attempt to establish a correspondence by matching a sparse sets of image features (usually edges). The number of points used is related to the number of image features identified. Although, feature-based algorithms work very fast, they can only generate sparse disparity maps. So they are not suitable for many applications (e.g. reconstructing surfaces) that require dense disparity maps.

Area-based algorithms [5][24] are also called correlation-based algorithms. These methods merge the feature detection step with the matching part, which means it deal with the images without attempting to detect salient objects in the images. Correlation-based stereo methods match neighbouring pixel values, within a window, between images.

The Multiple Scale Signal Matching (MSSM) [18] algorithm, which is a local correlation-based algorithm, is extended in our paper. Most of previous matching algorithms are applied on rectified stereo images with small image sizes (e.g. 640×480, 320×240), but our parallel algorithm can be applied to large un-rectified images (e.g. 4928×3264). The serial MSSM algorithm provided accurate disparity maps [15], but for the current application, it was no longer adequate due to its slow processing time. For the original Java version on a single core CPU, it takes around 20 minutes to process a single pair of 16MegaPixel images [2].

B. Multi-core CPU and GPU acceleration

Recent applications (e.g. mobile robots and autonomous vehicles) require fast stereo capture. A single core scalar CPU is too slow to give high resolution stereo with modern cameras. This has motivated research into parallelising the problem. Zhang et. al [26] proposed two parallel SIFT (Scale Invariant Feature Transform) algorithms with optimization techniques to improve the application's performance on multi-core systems. Jang et. al [8] implemented neural networks-based text detection system on both GPU and multi-core CPU processors. The paper of Yang et al. [22] was the first one to explore the potential of GPUs to accelerate depth estimation. They effectively used the capability of graphics hardware to warp and process images. Zhang et. al [25] applied a parallel Motion Estimation algorithm on a heterogeneous computing system, and compared its performance on a CPU and a GPU.

Researchers in stereo matching area also improved the algorithm's efficiency in recent years. Wang [20] achieved high quality results while maintaining real-time performance, using an adaptive aggregation step in a dynamic-programming stereo framework on GPU. Mei [13] also presented a GPU-based matching system that could effectively handle various errors in a multi-step refinement process, which gave good performance in both accuracy and speed. Mattoccia [12] took advantage of coarse-grained thread-level parallelism on multi-core CPU, which dramatically reduced the execution time.

In this paper, we discuss the Parallel Pyramid Matcher (PPM) [2] with various optimizations for multi-core CPU and GPU architectures, which improve the speed of MSSM algorithm dramatically. To the best of our knowledge, no one has shown how matching algorithm performances scale with different number of CPU cores, so we analyse the performance trend on several multi-core CPUs and then compare it with GPU performance. We also break down the stereo matching algorithm into several atomic sub-algorithms to further understand its performance acceleration.

III. Parallel Pyramid Matcher

In this section, we first introduce the stereo matching algorithm which we implemented (Sec. III.A), followed by brief explanations of fundamentals behind our method of acceleration of the algorithm on both CPUs (Sec. III.B) and GPUs (Sec. III.C).

A. Pyramidal Stereo Matching Algorithm

The initial idea of this matcher comes from the Multiple Scale Signal Matching (MSSM) algorithm [18], a correlation-based matching algorithm.

The aim of the Stereo Matching algorithm is to compute a disparity map for a stereo-pair of images. The disparity map refers a two dimensional array of displacement vectors which map the pixels in one image onto their corresponding pixel in the other. In the case of un-rectified images, i.e. images without lens distortion correction, there are two disparity maps,

one for horizontal displacements and one for vertical displacements, specifying orthogonal components of the disparities. The algorithm also produces a confidence map, which assigns a confidence to each disparity vector. The confidence map is a measure of the likelihood that the found correspondence is the best match for the pixel. In general, only pixels with high confidences lead to reliable results.

1) Pyramid Creation: The matching algorithm employs a pyramid representation (Figure 1), which is applied to the input images and then serves to facilitate signal matching at multiple scales [17]. In this scheme, an initial estimate for the disparity is computed at a low resolution and the initial disparity estimate from this scale is refined at higher resolutions until the target resolution is achieved.

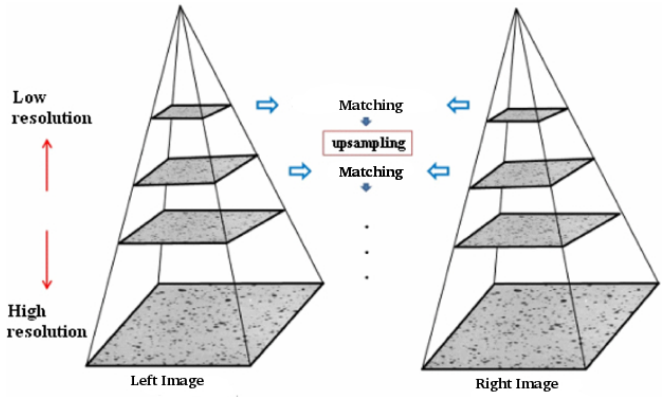

Fig. 1: Pyramid representation of stereo input image to perform matching at multiple scales

2) Pixel Correlation: In order to refine the disparities at each level of the input pyramids, the algorithm attempts to maximize the correlation between pixels in windowed regions of the left and right images. The windowed correlation is weighted by doing a convolution with a Gaussian kernel, which is computed as follows

$$cor_{l,r} = \frac{cov_{l,r}(x,y)}{\sqrt{var_l(x,y)}\sqrt{var_r(x,y)}} \tag{1}$$

where the covariance is computed as

$$cov_{l,r}(x,y) = \sum_u \sum_v f_l(x+u, y+v) f_r(x+u, y+v) w(u,v) \tag{2}$$

and the variance is computed as

$$var_i(x,y) = \sum_u \sum_v f_i(x+u, y+v) f_i(x+u, y+v) w(u,v) \tag{3}$$

In the above equations, u and v define the size of the window and $w(u,v)$ define the Gaussian weight. The window in the left image is named as reference window and the window in the right image is called search window. By moving the search window, a local correlation surface is created for each pixel position. The search window is moved in four directions (up, down, left and right) using a one pixel search step. If we include the null move, there are in total five local correlation maps.

3) Polynomial Maximization: Having obtained five correlation coefficient matrices, 2nd order polynomial maximization is applied to the corresponding elements of the matrices. The task is to find, for each pixel, the local maximum of the curve. If the local maximum is found to be more than one pixel away from the current position the relative displacement is clipped to ± 1. The confidence map, is computed from the local correlations after the move has been done.

4) Inter-Scale Disparity Refinement: At each scale-level, the disparities are anisotropically diffused using the confidence map. The effect is that disparities found in areas of high confidence tend to diffuse into adjacent areas of low confidence. Suppose $InitConf$ is the confidence before search. $InitDisp_{xy}$ is the disparity and $ConfHV$ is the confidence after search. Then

$$InitConf = ConfHV + 0.75 * (InitConf - ConfHV) \tag{4}$$

The disparity matrices are then weighted by the confidence matrix as the equation below.

$$\begin{aligned} Disp_{xy} &= \sum_{I \in \Delta} InitDisp_{(xy+I)} \times InitConf_{(xy+I)} \\ \Delta &= \{[0,0],[0,1],[0,-1],[1,0],[-1,0]\} \end{aligned} \tag{5}$$

In the above equation, $Disp_{xy}$ is the output disparity. This smoothing process is iterated for a certain number of times to refine disparity map.

5) Rescale: From current scale to higher resolution scale, interpolation is necessary for each pixel in the disparity map. An interpolated pixel is derived from its four neighbours. If

$$x_0 = floor(x) \quad y_0 = floor(y) \tag{6}$$

$$a = x - x_0 \quad b = y - y_0 \tag{7}$$

then

$$\begin{aligned} f_{interpo}(x,y) = (1-a)(1-b) f(x_0, y_0) + (1-a) b f(x_0, y_0+1) \\ + a(1-b) f(x_0+1, y_0) + a \cdot b \cdot f(x_0+1, y_0+1) \end{aligned} \tag{8}$$

The interpolated result is the initial disparity map for the next higher scale resolution.

Iteratively implementing the same matching algorithm on each scale until the highest resolution scale, the final and more accurate disparity map is produced.

Although the algorithm is $\mathbf{O}(n \log n)$ of the number of pixels in the image, the constant of proportionality is relatively large. For a pair of very high resolution color images (e.g. 4928×3264), we find that it takes of the order of 20 minutes when implemented in sequential Java. In order to accelerate the processing time of the algorithm, parallel programming (Multi-core CPU and GPU programming) was tried.

B. Vector Pascal Multi-core CPU Parallel Theory

The matching algorithm is first implemented on multi-core CPU by Vector Pascal. Vector Pascal [3][4], an open source compiler for Pascal, is designed to support efficient expression of algorithms using the SIMD (Single Instruction, Multiple Data) model of computation. It is a dialect of Pascal designed to make efficient use of the multi-media instruction

sets of recent processors. It supports data parallel operations and saturated arithmetic.

In Vector Pascal, all operators are overloaded, so that they can operate on arrays and vectors as well as scalars. Using compiler flags, a single program can be compiled, with differing levels of parallelism, to target a range of microprocessors. The Vector Pascal compiler uses pthreads to support multi-core parallelism over matrices. The default setting of task scheduling is using pthread semaphores, but another option, spin-locks (busy waiting), can be used to for machines with very large numbers of cores. In our experiments, all runs were done using spin-locks.

A semaphore is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a parallel programming or a multi user environment. It has a counter and will allow itself being acquired by one or several threads, depending on what value being posted and what its maximum allowable value is. If a semaphore cannot be acquired, it blocks, giving up CPU time to a different thread that is ready to run. This means that a few milliseconds pass before the thread is scheduled again.

Unlike semaphores, a spin-lock is a lock which causes a thread trying to acquire it to simply wait in a loop while repeatedly checking if the lock is available. Once acquired, spin-locks will usually be held until they are explicitly released. Although they avoid overhead from operating system process re-scheduling or context switching, spin-locks are efficient if threads are only likely to be blocked for a short period, since they prevent the core in question from being used by any other process.

C. GPU Parallel Strategies

Besides the implementation of the matching algorithm on multi-core CPUs, the algorithm was also implemented on a GPU to exploit its facilities for parallel computation. One of the most popular GPU Architectures is CUDA (Compute Unified Device Architecture), which is the software architecture of NVIDIA GPGPUs (General-Purpose Computing on Graphics Processing Units).

As can be seen in Figure 2, a thread is the fundamental unit of parallel programming. A thread block is a batch of threads that can cooperate with each other by synchronizing their execution, efficiently sharing data through shared memory, with the restriction that two threads from different blocks cannot cooperate. Each Grid contains several thread blocks and a kernel is executed as a grid of thread blocks. All threads in a block execute the same thread program. They have thread ID numbers within their block and the thread program uses its thread ID to select work and address shared data.

In this work, parallelisation consists of distributing the above-described stereo matching algorithm over three types of memories on a GPU (shared memory, global memory and texture memory) and pinned memory on CPU. Each type of memory and its subsequent parallelisation is discussed in the following sections.

1) Shared Memory: GPU Shared memory is fast and is shared between all streaming processors in a multiprocessor. A streaming processor is a fully pipelined, single-issue, in-order

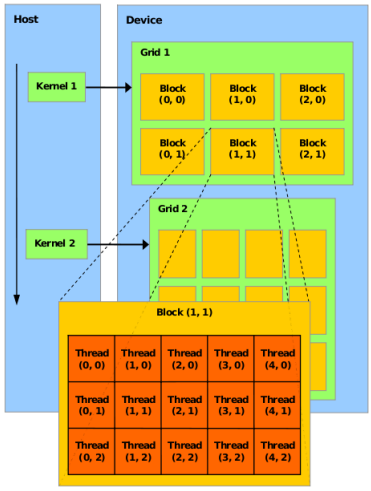

Fig. 2: CUDA Structure

microprocessor, and several streaming processors make up a streaming multiprocessor. To use shared memory, the data is first copied from host memory to global memory on the GPU device, and then indexed by block and thread ID. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. A single streaming processor only executes one thread program at a time, but all streaming processors in all multiprocessor work simultaneous, which results in a time effective execution. Finally, the data will be copied back to the host after execution on the GPU.

To accelerate our matching algorithm, for example, convolution procedures are implemented using shared memory. Convolution is used several times in the algorithm. It is used in step pyramid creation, pixel correlation and inter-scale disparity refinement. It is an important part for parallelisation optimisation and it demonstrates significant performance improvement.

2) Texture Memory: GPU Texture memory is available for reading by all multiprocessors. Data is fetched by texture units in a GPU. Textures are accessed through a dedicated read-only cache which is optimized for spatial locality, so the data can be interpolated linearly without extra overheads (see figure 3).

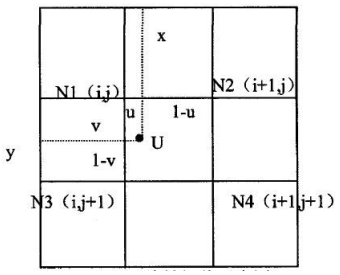

Fig. 3: Floating Point Interpolation

Suppose the four points N1, N2, N3 and N4 are known. In order to determine U, normally the equation below is needed,

which is a cost expensive calculation.

$$F(x,y) = (1-u)(1-v)F(i,j) + u(1-v)F(i+1,j) + (1-u)vF(i,j+1) + uvF(i+1,j+1) \quad (9)$$

On a GPU, this is implemented in hardware. The execution time cost for texture memory to perform linear floating point interpolation is negligible. Floating point interpolation is an important element in the matching algorithm. Polynomial maximization and rescaling image size both benefit from the fast access provided by the texture memory.

3) Global Memory: GPU Global memory is the largest volume of memory available to all multiprocessors in a GPU, but the latency is higher than shared memory. Global memory is mainly used to keep intermediate results for later use either by shared memory or by texture memory, so that reduces time spent on data exchange between CPU and GPU. Every step of our matching algorithm requires support from global memory.

4) Pinned Memory: CPU data is allocated on pageable host memory by default, where GPU cannot access data directly. When the data transfer from pageable host memory to device memory, the CUDA driver must first allocate a temporary pinned buffer, copy the host data to the pinned buffer, and then transfer the data from the pinned buffer to device memory [7]. This consumes precious host time, but the cost of the transfer between pageable and pinned host arrays can be avoided by directly allocating host arrays in pinned memory in CUDA C/C++.

IV. Experiments

In this section, we first describe the experimental setups (Sec. IV.A), i.e. the systems and data used in our experiments. Then we discuss our findings on accelerating the matching algorithm on CPUs and on the GPU (Sec. IV.B). We finally analyze the acceleration impact on CPU and GPU (Sec. IV.C).

A. Experimental Setup

The experiments have been conducted on three different multi-core CPU systems and one GPU. The CPU systems are: a four chip AMD Opteron processor 6366HE system, a two chip Intel Xeon processor E5-2440 system and a single chip system using an Intel Xeon processor E5-2620. The GPU model was a Nvidia GeForce GTX 770 (See Table I for details).

The test data for the matching algorithm originates from a stereo-pair of cameras of the binocular robot-head used for CloPeMa. The head is fitted with Nikon DSLR D5100 cameras that are capable of capturing images at 16 mega pixels [2]. The dataset consists of a set of stereo-pairs of images, covering a range of different cloth textures. These images were taken as full high resolution colour images with 4928 by 3264 pixels. In the system, the images are represented as arrays of 32 bit floating point numbers to maintain accuracy.

B. Parallel Stereo Matching Performance

Figure 4 shows the overall performance of the parallelised pyramid matching algorithm on three multi-core CPUs specified in Table I, with the number of threads varied from the minimum to the maximum of a given system. The x-axis corresponds to the number of threads, while the y-axis is the execution time in seconds. The axes are shown in log scale in order to present the figure more concisely. Note that all the CPUs support AVX instructions and the code was compiled to use these instructions. Since these instructions allow 8 fold SIMD parallelisation of vector operations, the single core performance is already substantially faster than the scalar Java performance. For fare comparison, single core performance without SIMD is shown in figure 4, which uses X87 instructions.

TABLE I: Specification of different processors (S for per socket, C for per core) used in this work

Parameter	AMD Opteron 6366HE	Intel Xeon E5-2440	Intel Xeon E5-2620	Nvidia GeForce GTX 770
Chips	4	2	1	1
Cores and Threads	16 and 16 (S)	6 and 12 (S)	6 and 12	1536
Clock Speed	1.8GHz	2.4 GHz	2 GHz	1046 MHz
Memory Capacity	504GB	24GB	16 GB	4 GB
Memory Technology	DDR3	DDR3	DDR3	GDDR5
Memory Speed	1333 MHz	1333 MHz	1333 MHz	1753 MHz
Memory Data Width	4 × 64 bits (S)	2 × 64 bits (S)	64 bits	256 bits
Peak Memory Bandwidth	51.2GB/s (S)	32 GB/s (S)	42.6 GB/s	224.3 GB/s
Data Caches	16 x 16 KB L1 (S) 8 x 2 MB L2 (S) 2 x 8 MB L3 (S)	384 KB L1 (C) 1.50 MB L2 (C) 15 MB L3 (S)	32 KB L1 (C) 256 KB L2 (C) 15 MB L3	64 KB L1 per SMX 512 KB L2

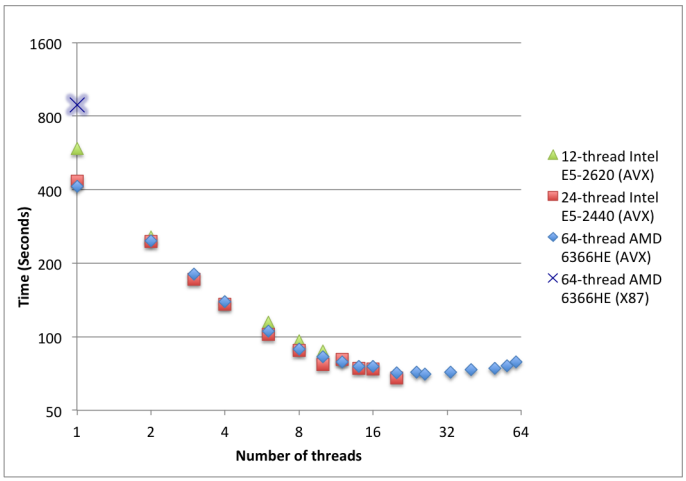

Fig. 4: Log/Log plot of matching performance on different CPU based machines

The performance of three machines show a similar trend. Not surprisingly, as the number of threads increased, execution time decreased. In the initial range, up to 10 threads the performance follows an approximately linear trend in log log space, which is equivalent to a power-law functional form. However, one interesting finding for the 64 cores AMD is, that after around 30 threads, the running time stopped decreasing, and increased slightly. This trend could not be verified on the other two machines, since the lower number of cores did not allow us to detect whether a similar turning point exists.

This scenario can not simply be explained by Amdahl's Law [6]. Amdahl's law states that if P is the proportion of a program that can be parallelised, and $(1-P)$ is the proportion that cannot be parallelised, the maximum speed-up that can be achieved using N processors is

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{10}$$

In the limit, as N tends to be infinity, the maximum speed-up tends to be $1/(1-P)$.

Theoretically, if there are enough cores (threads) to run an algorithm, the execution time of the part that can be parallelised will tend to zero. So in theory, the optimal performance is predictable, because it only depends on the serial part. But in practice, in the multi-core environment, the CPU takes time to assign different tasks to different cores. As the number of used cores grows, the allocation time increases, eventually the extra thread scheduling time can be expected to outweigh the speed-up due to parallelisation.

However, it is not clear that this is necessarily the cause of the bottoming out of run time at the 30 threads point. Another possible cause relates to memory bandwidth. Depending on how the mapping of physical to virtual memory is carried out, the images being processed may have been spread over the memory channels of all 4 processor chips on the Opteron system, or may have been entirely allocated within the memory controlled by one of the chips. In the latter case the bottoming out of performance after 30 threads could represent the saturation of the memory bandwidth of that chip. The input and output pyramids occupy around 2.5 GB in total. In addition, there are in the CPU version, various temporary buffers of comparable sizes that have to be manipulated repeatedly as the algorithm iterates. It thus places considerable demands on the memory channels.

Since similar results were observed for up to 30 threads across all three machines and only the 64-core AMD machine showed a performance bottleneck, the tests for the key sub-parts of the algorithm were only performed on the 64-core system.

As mentioned in Section 3, the matching algorithm can be separated into several parts: pyramid creation, pixel correlation, polynomial maximization, inter-scale disparity refinement, rescale and others. Execution time for each of the six parts over different number of cores (threads) is shown in Figure 5. The figure illustrates that as the number of cores varies, different parts show different performance changes. The two outstanding parts in Figure 5 are pixel correlation and polynomial maximization. Those two parts have similar execution times for single thread execution. However, when executed with multiple-threads, there are two major differences that can be observed: (1). the acceleration rate of polynomial maximization continues to improve until about 40 cores while the speed-up of pixel correlation reaches a peak and remains stable after around 12 cores; (2). the acceleration rate of polynomial maximization is much faster than pixel correlation. These findings are discussed in more details in the coming subsection on acceleration impact analysis.

The same matching algorithm and key sub-parts were also implemented for, and run on, the GPU. The GPU performances

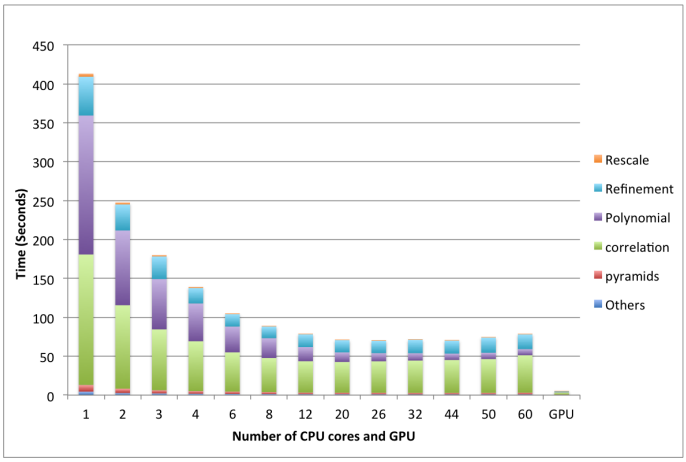

Fig. 5: Performance of matching algorithm breakdown on AMD 6366HE CPU and GPU

are shown in Figure 5, for comparison with the CPU results. Table II gives in detail, single-thread AMD CPU performance (using both AVX and x87 instructions), best AMD CPU performance (using 44 threads with AVX instructions) and GPU performance for each of the six key sub-parts. Besides the execution time for each sub-part, the table provides the corresponding speed-up of the AMD single-thread SIMD, best AMD CPU and GPU performance over the single-thread AMD CPU performance with X87 instructions. In this experiment, single-thread CPU means single-thread AMD CPU, and 44-thread represents best AMD CPU performance. It should be noted that single threaded AVX timing benefits from SIMD parallelisation - which is potentially a factor of 8 compared with using X87 code.

TABLE II: Single-thread AMD CPU with X87 and AVX instructions, 44-thread AMD CPU and GPU Performance

	CPU					GPU	
	Single-thread			44-thread			
	X87	AVX					
	Time(s)	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup
Pyramids	19.94	8.61	2.32	1.68	11.87	0.43	45.88
Correlation	486.56	167.69	2.90	42.34	11.49	2.90	167.89
Polynomial	192.19	178.51	1.08	8.18	23.50	0.26	731.12
Refinement	174.93	49.93	3.50	16.80	10.41	1.04	168.65
Rescale	5.57	3.91	1.42	0.39	14.28	0.19	29.89
Other tasks	7.95	4.46	1.78	1.16	6.85	0.20	39.86
Total	887.14	413.11	2.15	70.55	12.57	5.02	176.77

Overall, CPU gives about $12\times$ and GPU gives $176\times$ acceleration. Among the sub-tasks, polynomial maximization gives the best parallel performance with 23 times and 731 times speed-up on CPU and GPU respectively. This process is full of array calculations, and every single step can be parallelised. This type of process suits well for parallelisation with a multi-core architecture. Simple linear interpolation, which is an important and basic step in the matching algorithm, also has this characteristic.

On the other hand, the most time consuming part of the algorithm is pixel correlation. The best CPU performance reaches a 11 times speed-up, but gives only 4-fold speed-up compared with single thread AVX. This section contains

many iterations of image convolution, another basic image algorithm. Hence, the behaviour of convolution is the key to understanding that of pixel correlation.

From the performance table, we notice that the speed-up for polynomial maximization on a GPU is much higher than for other components. This is due to the polynomial maximization being the only part which runs entirely on the GPU without any data transfer to CPU. In the next sub-section, it is demonstrated how much data transfer between the CPU and GPU can influence the overall performance.

C. Acceleration Impact Analysis

In Figure 5, it has been shown that polynomial maximization and pixel correlation exhibit different acceleration rates. Linear interpolation and image convolution are used in different proportions in these two parts of the algorithm. Since we want to understand the underlying reasons for the differences in acceleration, linear interpolation and image convolution were examined in more detail. These operations are much simpler, allowing for a more refined performance analysis.

1) Acceleration on CPU: Figure 6 shows the linear interpolation and image convolution performance on different number of cores. Y axis is scaled by *Log* function. As the number of cores increases, both execution times decrease. The acceleration of interpolation demonstrates a good, power

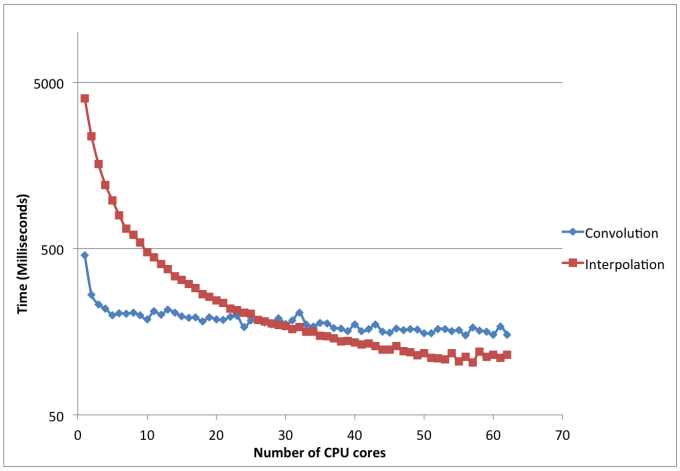

Fig. 6: Interpolation and convolution performance on different number of CPU (AMD 6366HE) cores using AVX instructions

law like, curve with a performance improvement of 36 times. Linear interpolation only contains one step of array access and is easily parallelised on a multi-core architecture. Polynomial maximization and rescale, which rely on linear interpolation, follow therefore a similar speed-up trend.

The performance trend for convolution, on the other hand, is similar to the trends for overall matching performance and pixel correlation performance. Execution time decreases as the number of cores increases, until about 10 cores are reached. After that, the execution time decreases very slowly. It only gives about 3 times speed-up on CPU, which is close to pixel correlation performance. Note that all these tests are using AVX instructions.

A possible explanation is that to perform image convolution, a one dimensional convolution is first applied horizontally and then vertically to the image. Although this method allows straight forward parallelisation, the creation of an intermediate result array will limit the speed-up gain since the image data size is typically larger than the CPU cache. Writing to the intermediate array will tend to flush the original data buffer from the caches.

We can observe that although the interpolation is slower than convolution on a single-core CPU, as the number of cores increases, interpolation overtakes convolution. From the previously described experiment, it is understood that different algorithms impose different loads on the memory subsystems. This appears to be the reason as to why the different algorithms show different trends. To conclude, compared to interpolation, although two pass convolution still benefits from multi-core parallelisation, this type of convolution algorithm is not well suited to effective parallel acceleration.

2) Acceleration on GPU: Interpolation and image convolution performance on GPU is presented in Figure 7. On the

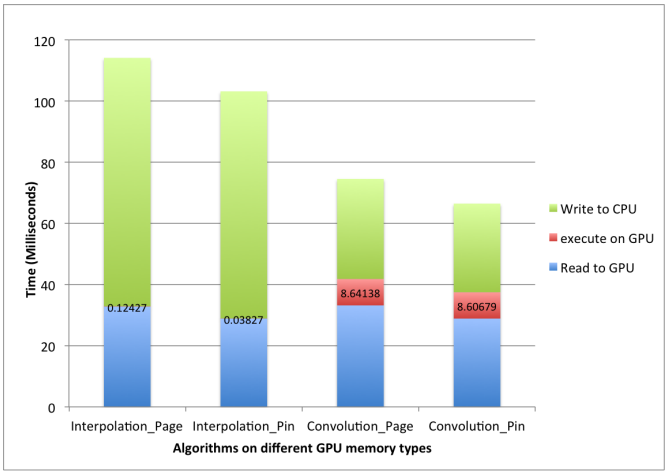

Fig. 7: Interpolation performance on GPU

GPU, interpolation gives about 39 times speed-up compared to a single-core CPU using AVX instructions. It is not as big of an improvement as on polynomial maximization, but after breaking down the interpolation process on GPU, it can be seen that almost all the time is consumed on data transfer between the CPU and GPU. Although, using pinned memory, rather than paged memory, to reduce the transfer time it is still the most time consuming part. Comparatively speaking, the execution time on the GPU is minimal. This is due to one of the main advantages of the GPU. The GPU has texture memory facilitating the interpolation. As the size of image increases, there is no significant execution time increment shown on the GPU, which demonstrates that doing floating point interpolation on texture memory is almost cost-free, as long as the image data can fit into texture memory.

With convolution, the GPU gives about a 7 times speed-up over a single-core CPU using AVX. Compared to interpolation, this is only a marginal improvement. Breaking down the convolution process, the most time consuming part is again the transfer time. However, for convolution, execution time on the

GPU is short but nonnegligible. The main function of the convolution, executes on the shared memory, which is extremely fast. But similar to the CPU implementation, the algorithm generates intermediate variables in global memory. Therefore, data transfer within the GPU memories is inevitable. This accounts for most of the execution time on the GPU.

Both GPU versions of interpolation and convolution give significant acceleration improvements compared to CPU performance. Although as an isolated algorithm, convolution's performance is bounded by CPU to GPU transfers, when operating as a component of the matcher it is invoked repeatedly using data that has already been transferred to the GPU. This would explain why GPU gives 176 times speed-up on the complete matching algorithm.

This scenario suggests, when implementing algorithms on GPU, one should let GPU do parallel computing as much as possible, and try the best to keep intermediate variables on GPU only, because the data transfer within GPU is much faster than transfer to CPU (see Table I).

V. Conclusion

We demonstrated the successful implementation of a parallel pyramid matcher, i.e. the dense-correspondence matching algorithm for multi-core CPUs and GPU. The experimental results showed for both platforms significant performance accelerations with multi-core CPUs giving $12\times$ speedup and GPU giving $176\times$ speedup, compared to non-SIMD single-core CPU performance. In addition, we found that the optimised acceleration of interpolation has a greater impact on the overall speed-up than convolution as it is more suited for parallel execution, but convolution currently has a speed-up limitation due to it requiring storage of intermediate results.

In the future, we plan to accelerate a variety of stereo-matching algorithms on different parallel architectures, including other multi-core CPU, graphic processor and coprocessors (i.e Xeon Phi). By analysing the performance over different number of cores, we shall investigate which type of matching algorithm is more suitable for parallel computing to obtain better accelerations. Moreover, we intend to propose approximated stereo matching approaches for further accelerating the binocular robot-head. This may involve trading off accuracy for execution speed.

Acknowledgment

This work was partially funded by Clothes Perception and Manipulation (CloPeMa) EU project.

References

[1] Stan Birchfield and Carlo Tomasi. Multiway cut for stereo and motion with slanted surfaces. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 1, pages 489–495. IEEE, 1999.

[2] W. Paul Cockshott, Susanne Oehler, Tian Xu, J. Paul Siebert, and Gerardo Aragon-Camarasa. Parallel stereo vision algorithm. In *MARC@RWTH*, pages 45–50, 2012.

[3] W.Paul Cockshott, editor. *SIMD Programming Manual for Linux and Windows*. Springer, London, 2004.

[4] W.Paul Cockshott, editor. *Glasgow Pascal Compiler with vector extensions*. Glasgow, 2011.

[5] Olivier Faugeras, Thierry Viéville, Eric Theron, Jean Vuillemin, Bernard Hotz, Zhengyou Zhang, Laurent Moll, Patrice Bertin, Herve Mathieu, Pascal Fua, et al. Real-time correlation-based stereo: algorithm, implementations and applications. 1993.

[6] Amdahl Gene. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, number 30, pages 483–485, 1967.

[7] Mark Harris. How to optimize data transfers in cuda c/c++. *Technical report*, 2012.

[8] Honghoon Jang, Anjin Park, and Keechul Jung. Neural network implementation using cuda and openmp. In *Digital Image Computing: Techniques and Applications, 2008*, pages 155–161. IEEE, 2008.

[9] Andreas Klaus, Mario Sormann, and Konrad Karner. Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 3, pages 15–18. IEEE, 2006.

[10] Vladimir Kolmogorov and Ramin Zabih. Computing visual correspondence with occlusions using graph cuts. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 2, pages 508–515. IEEE, 2001.

[11] Jeremy Maitin-Shepard, Marco Cusumano-Towner, Jinna Lei, and Pieter Abbeel. Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2308–2315. IEEE, 2010.

[12] Stefano Mattoccia. Fast locally consistent dense stereo on multicore. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 69–76. IEEE, 2010.

[13] Xing Mei, Xun Sun, Mingcai Zhou, Haitao Wang, Xiaopeng Zhang, et al. On building an accurate stereo matching system on graphics hardware. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 467–474. IEEE, 2011.

[14] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3):7–42, 2002.

[15] JP Siebert and CW Urquhart. C3d^{TM}: a novel vision-based 3-d data acquisition system. In *Image Processing for Broadcast and Video Production*, pages 170–180. Springer, 1995.

[16] Federico Tombari, Stefano Mattoccia, and Luigi Di Stefano. Segmentation-based adaptive support for accurate stereo correspondence. In *Advances in Image and Video Technology*, pages 427–438. Springer, 2007.

[17] Arthur A. van Hoff. Efficient computation of gaussian pyramids. *Technical report*, 1992.

[18] Arthur A. van Hoff. An efficient implementation of mssm. *Technical report*, 1993.

[19] V Venkateswar and Rama Chellappa. Hierarchical stereo and motion correspondence using feature groupings. *International Journal of Computer Vision*, 15(3):245–269, 1995.

[20] Liang Wang, Miao Liao, Minglun Gong, Ruigang Yang, and David Nister. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *3D Data Processing, Visualization, and Transmission, International Symposium on*, pages 798–805. IEEE, 2006.

[21] Bryan Willimon, S Birchfield, and Ian Walker. Classification of clothing using interactive perception. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1862–1868. IEEE, 2011.

[22] Ruigang Yang, Greg Welch, and Gary Bishop. Real-time consensus-based scene reconstruction using commodity graphics hardware? In *Computer Graphics Forum*, volume 22, pages 207–216. Wiley Online Library, 2003.

[23] Kuk-Jin Yoon and In So Kweon. Adaptive support-weight approach for correspondence search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):650–656, 2006.

[24] Ramin Zabih and John Woodfill. Non-parametric local transforms for computing visual correspondence. In *Computer Vision ECCV'94*, pages 151–158. Springer, 1994.

[25] Jinglin Zhang, J-F Nezan, and J-G Cousin. Implementation of motion estimation based on heterogeneous parallel computing system with opencl. In *IEEE 14th International Conference on High Performance Computing and Communication*, pages 41–45. IEEE, 2012.

[26] Qi Zhang, Yurong Chen, Yimin Zhang, and Yinlong Xu. Sift implementation and optimization for multi-core systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.