Comparison of FPGA and GPU implementations of Real-time Stereo Vision

Ratheesh Kalarot and John Morris
Department of Computer Science
The University of Auckland, Auckland, New Zealand
rkal018@aucklanduni.ac.nz, j.morris@auckland.ac.nz

Abstract

Real-time stereo vision systems have many applications - from autonomous navigation for vehicles through surveillance to materials handling. Accurate scene interpretation depends on an ability to process high resolution images in real-time, but, although the calculations for stereo matching are basically simple, a practical system needs to evaluate at least 10^9 disparities every second - beyond the capability of a single processor. Stereo correspondence algorithms have high degrees of inherent parallelism and are thus good candidates for parallel implementations. In this paper, we compare the performance obtainable with an FPGA and a GPU to understand the trade-off between the flexibility but relatively low speed of an FPGA and the high speed and fixed architecture of the GPU. Our comparison highlights the relative strengths and limitations of the two systems. Our experiments show that, for a range of image sizes, the GPU manages 2×10^9 disparities per second, compared with 2.6×10^9 disparities per second for an FPGA.

1. Preamble

Current Field Programmable Gate Array (FPGA) technology provides thousands of small logic elements (called logic elements or logic blocks) embedded in a rich connection matrix. This permits arbitrary computation blocks to be built from the basic computing blocks: both pipelined and parallel circuits are readily constructed. However, the general purpose nature of the basic blocks mean that they are relatively slow compared to larger blocks built for specific purposes, e.g. the Arithmetic and Logic Units (ALU) of a Graphic Processing Unit (GPU). Also, although a wide variety of communication patterns can be programmed into a circuit, they are also relatively slow because there are many programmable switch points on most paths. On the other hand, GPUs provide several hundred ALUs coupled to small blocks of memory ($\sim 16kB$) and a large block of slower memory. They are designed for image rendering and the architecture is optimized for this task, but, the large number of state-of-the-art ALUs has generated much interest in using them for other computationally intensive tasks. Connection patterns are fixed and the challenge is often to move data in a way that allows the full computational power to be exploited.

Real-time stereo vision has numerous important applications, navigation for autonomous vehicles, surveillance, tracking people or objects in dynamic scenes and industrial control: most systems which attempt to replicate human capabilities need the 3D scene information that stereo can provide. In an artificial vision system, the distorted images captured by real cameras must first be corrected and aligned. The key task is then matching pixels from left and right cameras - the so-called *correspondence problem* to obtain depth by triangulation. The inherent parallelism and high computational requirements of stereo algorithms have made FPGAs and GPUs primary choices for practical implementations. However, despite advancements in design tools, exploiting parallelism both platforms still requires significant effort.

1.1. Related work

Several FPGA based stereo vision implementations have been described: we have estimated the overall performance in disparities computed per second (disp/sec) achieved by each implementation. Darabiha *et al.*'s phase-correlation stereo matching runs at 30fps($0.06 \times 10^9 disp/sec$) on 320×240 images [1]. Park and Jeong [2] reported similar performance ($0.3 \times 10^9 disp/sec$) with a dynamic programming algorithm. Our implementation [3] processes 1024×768 images at 30fps($2.6 \times 10^9 disp/sec$).

Earlier GPU efforts tweaked code for the specialized GPU functional units (*e.g.* vertex shader and pixel shader) [4, 5]. Programming flexibility was a real concern in these cases. Belief propagation and graph cut have been implemented using nVidia's CUDA [6, 7]. Wang *et al.*[8]($0.06 \times 10^9 disp/sec$) and Gong *et al.*[9] implemented dynamic programming stereo for low disparity ranges.

Asano *et al.*[10] compared FPGA($22 \times 10^9 disp/sec$) and GPU implementations of a local block matching algorithm.

They reported very poor GPU performance, even worse than a multi core general purpose computer, which they attributed to memory access conflicts.

High resolution, strictly real-time *systems* have received less attention. Here we discuss a Symmetric Dynamic Programming Stereo (SDPS) algorithm [11] implementation on both FPGA and GPU. Both systems provide end to end stereo capability including lens distortion removal, rectification and produce identical depth maps.

2. Real-time Stereo

Many algorithms have been proposed which have varying trade-offs between matching quality, speed and resources needed. Dynamic programming algorithms lie in the middle of the spectrum, with reasonably matching performance and high speed - at the expense of relatively large memory for the predecessor array.

2.1. Symmetric Dynamic Programming Stereo

SDPS is described in full elsewhere [11, 3]; its key distinguishing feature is that, rather than attempt to reconstruct depths for the left or right image, it constructs a Cyclopæan image - one that would be seen by a virtual camera placed midway between the optical centres of the two real cameras. A typical dynamic programming algorithm, it accumulates costs for a best path through disparity space along a single scan line. Thus it only averages noise in a single direction - along a scan line - in contrast to global algorithms, *e.g.* belief propagation or graph cut [12, 13], which optimize over larger regions. As it works along the scan line, it stores the best predecessor for each disparity in the predecessor array - a large block of memory of size $\mathcal{O}(w\Delta)$, where a scanline has w pixels and there are Δ distinct possible disparities.

The brief outline below highlights the computations and storage needed, so that the constraints on the two implementations may be understood.

If $dI(x, d)$ is a signal dissimilarity measure between pixels at $x - \frac{d}{2}$ and $x + \frac{d}{2}$ in a scan line, SDPS accumulates costs, $C_{x,d,vs}$, for each disparity value, d, for possible visibility states, $vs \in \{B - binocularly\ visible, ML - monocular\ left, MR - monocular\ right\}$ at pixel, x, in the Cyclopæan image.

The costs to reach the three visibility states are:

$$C_{x,d,B} = dI(x,d) + \\ min(C_{x-1,d-1,ML}, C_{x-2,d,B}, C_{x-2,d,MR}) \quad (1)$$

$$C_{x,d,ML} = occ\_cost + \\ min(C_{x-1,d-1,ML}, C_{x-2,d,B}, C_{x-2,d,MR}) \quad (2)$$

$$C_{x,d,MR} = occ\_cost + \\ min(C_{x-1,d+1,MR}, C_{x-1,d+1,B}) \quad (3)$$

where *occ_cost* is a constant cost associated with a disparity change which acts as smoothness constraint. The predecessors, $\pi_{x,d,s}$, are :

$$\pi_{x,d,B} = argmin(C_{x-1,d-1,ML}, C_{x-2,d,B}, C_{x-2,d,MR}) \quad (4)$$

$$\pi_{x,d,ML} = argmin(C_{x-1,d-1,ML}, C_{x-2,d,B}, C_{x-2,d,MR}) \quad (5)$$

$$\pi_{x,d,MR} = argmin(C_{x-1,d+1,MR}, C_{x-1,d+1,B}) \quad (6)$$

Visibility constraints on transitions beween states mean that they are the only data needed in the predecessor array [11]. The back-track module, knowing the allowed transitions, is able to reconstruct the disparity profile from a sequence of visibility states.

Parallel implementations of SDPS proceed in *even* and *odd* phases: *even*|*odd* phases compute even|odd disparity values. This is required by the dependencies in Equations 1 - 3: for example, for even d, $C_{x,d,B}$ depends on $C_{x-1,d-1,ML}$ which is calculated in the preceding odd phase. Thus $\Delta/2$ cost calculation circuit modules or threads can calculate even and odd disparities in sequence.

2.2. Latency

For a true real-time application, latency is a critical factor. At 30fps, the host, which will be responsible for image interpretation and strategy determination, will be sitting idle for $> 30ms$ while a naive GPU implementation acquires the first image. Therefore both our implementations have attempted to minimize latency by continuously streaming pixel data from the cameras through the rectification and correspondence blocks to the host, which is able to commence interpretation with a latency of some tens of microseconds rather than the tens of milliseconds that would be necessary if a whole frame delay was permitted.

3. FPGA Implementation

The FPGA implementation is shown in Figure 1; it consists of three blocks: (a) distortion removal and rectification (*cf.* Section 3.1), (b) disparity calculators (*cf.* Section 3.2) and (c) predecessor array and backtrack module (*cf.* Section 4.5).

3.1. Distortion removal and rectification

Real lenses distort images and it is difficult to align the cameras perfectly in the canonical configuration, so the 'raw' images produced by the cameras must be corrected. Our approach combines distortion removal and alignment correction into a single step by using lookup tables. This approach is simple and fast and has the further advantage that

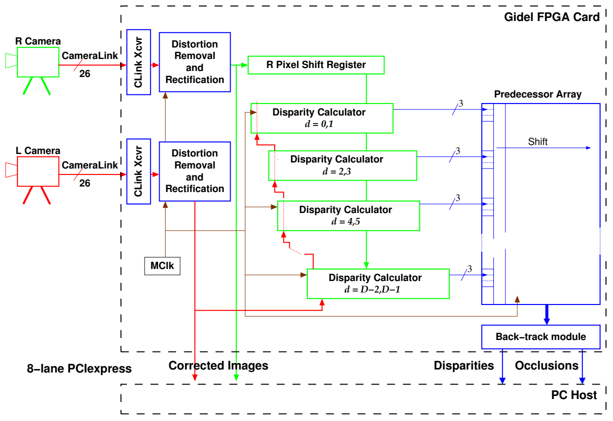


Figure 1. FPGA implementation: block diagram

no assumptions need be made about the nature of the distortion or misalignment - other than that the correction curves are smooth [14]. However, for large ($\gtrsim$ 1 Mpixel) images, the lookup tables required by the simplest approach (one lookup table entry per pixel) require more memory than is available in current state-of-the-art FPGAs. Thus we, relying on our assumption that the correction curves are smooth, reduced the lookup tables in a procedure which iteratively halved the size of the lookup table, then checked that the error introduced by interpolation from the reduced table was less than 1 intensity unit, until the table size could not be reduced further without introducing an unacceptable error in the pixel intensity. This dramatically reduces the lookup table size, for our laboratory system, a 65×65 entry lookup table suffices for 1024×768 pixel images.

3.2. Disparity calculators

One disparity calculator block is instantiated for every (odd,even) pair of disparities, $\frac{\Delta}{2}$ in all. It stores 6 costs, one for each visibility state for two disparity values, and updates them following Equations 1 - 3. As the new costs are evaluated, predecessor values, from Equations 4 - 6, are determined and despatched to the predecessor array. The disparity calculator is driven by a master clock with frequency, $f_{master} > 2f_{pixel}$ and is a four stage pipeline [3] to ensure that new costs can be calculated every $15ns$.

3.3. Minimum cost

At the end of each scan line, the minimum accumulated cost and its (disparity,state) index is captured by the back-track module. There are 3×128 candidates for minimum cost - so this is determined in a combining tree over 3 clock cycles.

3.4. Predecessor array and backtrack module

The predecessor array stores $\pi_{x,d,s}$ values that it receives from the disparity calculators. As it receives each new set, it pushes the previous values out to be read by the backtrack module: this makes efficient use of the limited memory available even in large state-of-the-art FPGAs.

The backtrack module starts with the index of the minimum path cost identified at the end of a scan line and, using the rules for allowed transitions, works backwards along a scan line, building a depth profile in reverse order. Note that it does this while the following scan line costs are being computed - introducing an inevitable scan line of latency but enabling high pixel clock rates to be supported.

Finally, the corrected left and right images, the depth map *and* the occlusion map are streamed to the host, which reads them with, typically, only a few tens of scanlines (or some tens of microseconds) of latency. Thus the host can begin to interpret the scene based on 3D data with a delay that is much shorter than a full frame time.

4. GPU Implementation

Our GPU implementation was based on an nVidia GeForce GTX 280 card connected to a host CPU via PCIexpress. The GTX 280 has 30 multiprocessors with a processor clock of 1296 MHz. It has 2GB global memory operating on a memory clock of 1107 MHz. CUDA was used to program the GPU.

4.1. Compute Unified Device Architecture (CUDA)

nVidia's Compute Unified Device Architecture (CUDA) provides a generic architecture for GPUs [15]; it offers a flexible programming model with minimal extensions to C. CUDA's view of a GPU has many Streaming Multiprocessors ($SM_0 \ldots SM_n$), where each multiprocessor consists of a set of Streaming Processors ($SP_0 \ldots SP_m$). Each multiprocessor has a fast local memory shared by its streaming processors, called the *shared memory*. All multiprocessors have access to a global or *device* memory. Each multiprocessor has a common instruction unit and very light weight thread scheduling mechanism allowing it to switch threads very quickly in contexts such as an I/O wait. A number of threads $t_0 \ldots t_n$ can be assigned to an SM forming a thread *block*. A block physically maps to an SM but the number of threads in a block can be more than number of SPs, in which case threads are split into warps so that only one warp is active at a time. Active threads execute the same instruction in parallel. If there is a branch in the code, threads agreeing on instructions execute in parallel while the other branches wait for them to complete.

To use GPU computing resources efficiently and to minimize costly device memory access, the components of the stereo system need to be carefully mapped to the GPU computing architecture. Two principles guided our mapping:

1. Avoid unnecessary device memory access and preferring shared memory for repeated memory access and

inter thread communication and

2. Avoid branches so as to keep all the processors doing useful work - we have improved on a previous version of this work [16] by combining even and odd calculations into a single thread, which in turn permits larger disparity ranges to be handled, see figures for $\Delta = 256$ in Table 1.

The shared memory available per multiprocessor in currently available GPUs is limited and most of them offer about 16KB shared memory per block.

4.2. Transfer of images to the GPU

Since we are unable to connect cameras directly to the GPU (as for the FPGA implementation (*cf.* Section 3), transfer of the images to and from the GPU via the PCIexpress bus needs to be scheduled carefully to reduce latency and ensure that the host starts to receive 3D data as soon as possible.
In our implementation, image data is transferred to the GPU's texture memory in blocks of lines. The number of lines is decided by two factors: the minimum number of lines needed to give some work to all the GPU's processors and a number of lines determined by the distortion and misalignment - rectification cannot start until this number of lines is available in the GPU. GPU texture memory has a cache with long lines which accelerate access to neighbouring pixels - a feature which is well suited to the access patterns in rectification. A benefit of the GPU texture memory is that it automatically clamps values outside the source image to the nearest edge pixel. It also contains interpolation circuitry which we use in the rectification procedure.
The lookup tables needed for rectification are fixed by the camera configuration and optics and are transferred once to GPU texture memory. Since the GPU is not as constrained for global memory as the FPGA, we used a full lookup table rather than the reduced ones used by the FPGA.

4.3. Rectification

Lookup tables ($Xmap$ and $Ymap$) are used - just as in the FPGA implementation - but full lookup tables allow removal of some interpolation computations performed by the FPGA. One multiprocessor is allocated to each scan line forming a block of threads. To optimize data flow to the forward pass (see Section 4.4), the number of threads per block is set to half the number of disparity values, $\frac{\Delta}{2}$. The rectification of a full scan line is divided into $\frac{\Delta}{2}$ sets of contiguous pixels mapping to threads $t_1, t_2 \dots t_{\frac{\Delta}{2}}$.
The latency introduced by the rectification stage is essentially the same as the FPGA implementation and is primarily determined by stereo configuration.
A synchronization point is inserted at the end of each rectification thread to ensure that all threads finish storing the

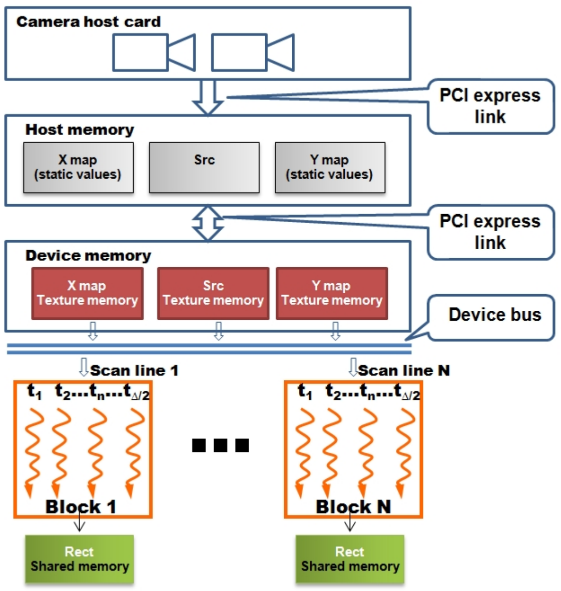


Figure 2. Rectification

rectified scan line segments in the shared memory array before proceeding to next step.

4.4. Forward pass

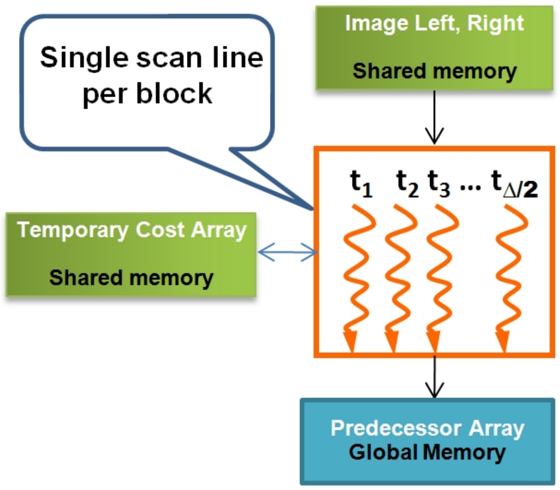

Single scan line per block
Image Left, Right Shared memory
t_1 t_2 t_3 … $t_{\Delta/2}$
Temporary Cost Array Shared memory
Predecessor Array Global Memory

Figure 3. Forward Pass: Δ logical threads run on each multiprocessor to process a single scan line

Figure 3 shows the mapping of SDPS' forward pass to the GPU. The input is the rectified scan line stored in shared memory of each SM. The forward pass uses the threads allocated in the previous stage. In one cycle, threads $t_1 \dots t_{\frac{\Delta}{2}}$ compute predecessors for all the even disparities in parallel followed by the odd disparities for one Cyclopæan pixel. Cyclopæan pixels are processed one by one in this fashion.

The even-odd separation and their synchronisation is ncessary as per the SDPS procedure. In this process, each thread accesses the cost array C in shared memory. Figure 4 shows the pseudo code for calculating predecessor array elements, π. This code is executed in each thread, $t_0 \dots t_{\frac{\Delta}{2}}$, in Figure 3 in parallel where each thread maps to one disparity pair. Since the computation of disparities only depends on two previous costs (one for even disparity and one for odd disparity), we only store the most recent cost for each disparity. From Equations 1 - 3, we see that the cost computation for the three visibility states (ML, B and MR) for a disparity, d, depends only on costs for a previous pixel, $x - 1$ and neighbouring disparities, $d \pm 1$. The $\frac{\Delta}{2}$ threads running on each multiprocessor will be swapped in and out as necessary and synchronization is required to make sure that all threads effectively run in lock-step. Odd-even sequencing in a single thread makes sure that preceding cost is updated before it is read. It easy to achieve this in CUDA as it has a built-in intrinsic for thread synchronization. This phase outputs the predecessor array, π, and final cost array C. Note that ML and B nodes always produce the same predecessor, labelled P1, and MR nodes another labelled P2. The predecessor array needs to be saved in device memory as it will not fit in the shared memory. In our implementation, we cache predecessor array elements for multiple Cyclopæan pixels and compact them into 128 byte blocks before writing to device memory. Latency in the forward pass depends upon the computation time and the number of scan lines processed in parallel: this is determined by the number of multi processors available on the target GPU. Since we have 30 multiprocessors, 30 scan lines can be processed at a time to ensure that all processors are working. Thus the latency is at least the processing time for 30 scan lines. This can increase with poor optics and poor mechanical alignment due to buffering of scanlines in the rectification module.

```
For each cyclopean pixel x
{
  for all even disparities in parallel
  {
    Update cost array ;
    Save predecessor array
    element in global memory;
  }
  Synchronise all threads;
  for all odd disparities in parallel
  {
    Update cost array ;
    Save predecessor array
    element in global memory;
  }
}
```

Figure 4. Pseudo-code for forward pass

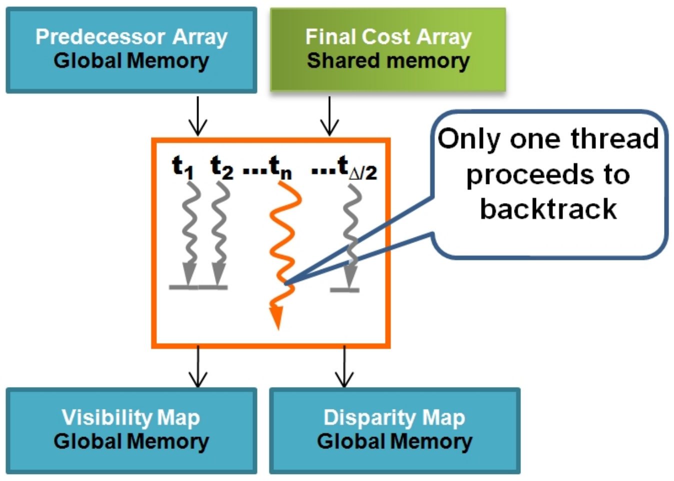


Figure 5. Backtracking: Note that the final cost array can be in (fast) shared memory.

4.5. Back track

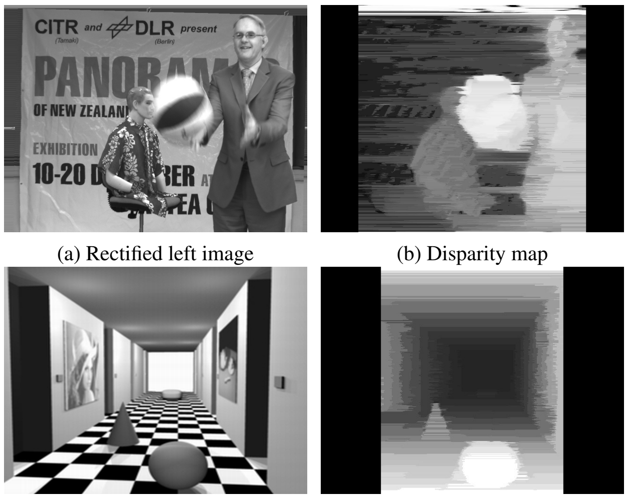
Once the index (disparity, visibility state) of the minimum cost has been determined, a single thread builds the depth profile backwards by reading predecessors from π, see Figure 5. Although an independent set of threads could back track several scan lines in parallel making more effective use of the processors [16], this would require the cost array to be transferred to the *device* memory. There is a tradeoff between latency and parallelism here: to use more processors, we would have to wait until sufficient scan lines had been transferred.

4.6. Performance

Example depth map results from the FPGA hardware in our laboratory are shown in Figure 6b and for the GPU processing the well known 'Corridor' images [17] in Figure 6d. The algorithm used is identical and will produce the same results from both implementations for the same input.
Table 1 shows relative performance of the two systems. A common misconception is that GPU implementations are universally handicapped by bus transfer throughput: in this case, this adds less than 2% to the total time. Table 2 shows the breakdown of GPU processing time for a typical configuration. The major portion of the time is spent on the forward pass where the different combinations of pixels are matched.

5. Conclusions

For this application at least, the FPGA implementation is superior, despite a much slower internal clock. It overcomes



(a) Rectified left image (b) Disparity map

(c) Left image (d) Disparity map

Figure 6. Resulting depth maps from SDPS on sample images

Image Resolution	GPU GTX 280		FPGA Altera Stratix III	
	fps	disp/sec $\times 10^{-9}$	*fps*	disp/sec $\times 10^{-9}$
Disparity range, $\Delta = 128$				
1024×768	20	1.76	30	2.6
640×480	55	1.74	80*	2.5*
320×240	230	1.4	400*	2.4*
256×256	310	1.3	450*	2.2*
Disparity range, $\Delta = 256$				
2048×1536	2.86	2.02	*Not feasible on FPGA*	

Table 1. Performance - GPU *vs* FPGA. All results are for 8 bit grey images. The FPGA was configured for 1024×768 images (locked to the capability of the Sentech FC83 CameraLink cameras used) whereas the GPU can handle several image sizes. Starred(*) results for the FPGA were estimated by de-rating 1024×768 pixel results to allow for the increased overhead of re-trace times and blank lines in typical cameras. Note that the GPU runs ~ 50 times faster than the same algorithm on a CPU (64 bit 2.33GHz Intel Xeon) using a single core

Operation	Time used(%)
Rectification	1.9
Forward Pass	93.4
Back tracking	3.6
Host to GPU memcopy	0.6
GPU to host memcopy	0.5

Table 2. Breakdown of GPU time: 1024×768 image at disparity range, $\Delta = 128$.

the limitation of its slow clock with extensive pipelining - it does not need to wait for intermediate results to be aggregated - they are passed immediately to succeeding pipeline stages. The GPU has significant *internal* overheads which slow it down: for this application, the I/O transfer time (time to transfer an image across the bus) is not the problem - it was measured and found to contribute less than 2% of the total time.

However, space limitations on cost effective FPGAs make it difficult to fit circuits that handle larger disparity ranges into all but the largest currently available FPGAs - the FPGA used in our laboratory has a similar cost to the high end GPU used. The GPU is able to handle a disparity range of 256 (*vs* 128 for the FPGA) with a concomitant increase in the disparities calculated per second: the larger disparity range provides more trivially exploited parallelism for the GPU.

The FPGA implementation also uses a custom circuit to back-track in parallel with cost computation for the succeeding line: it can do this because the FPGA memory is accessed by both cost computation modules and the back-track module. Further, it is used as a FIFO with a read pointer moving just ahead of the write pointer. In contrast, the GPU only makes use of a single thread for backtracking to save some costly memory transfers. The ability to implement efficient memory with different access semantics (random access, FIFO, stack *etc.*) is a key difference between the two architectures.

Finally, we note that almost all parallel algorithm implementations depend on relatively high computation:communication cost ratios. Among the stereo algorithms, SDPS has a higher ratio than most algorithms (particularly the belief propagation family) making it relatively easy to obtain good speed-ups.

References

[1] A. Darabiha, J. Rose, and W. J. MacLean, "Video-rate stereo depth measurement on programmable hardware," in *CVPR*. IEEE Computer Society, 2003, pp. 203–210. [Online]. Available: http://csdl.computer.org/comp/proceedings/cvpr/2003/1900/01/190010203abs.htm 1

[2] S. Park and H. Jeong, "Real-time stereo vision fpga chip with low error rate," in *MUE '07: Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 751–756. 1

[3] J. Morris, K. Jawed, G. Gimel'farb, and T. Khan, "Breaking the 'ton': Achieving 1% depth accuracy from stereo in real time," in *Image and Vision Computing NZ*, D. Bailey, Ed. IEEE CS Press, 2009. 1, 2, 3

[4] Q. X. Yang, L. Wang, and R. G. Yang, "Real-time global stereo matching using hierarchical belief

propagation," in *BMVC*, 2006, p. III:989. [Online]. Available: http://www.bmva.ac.uk/bmvc/2006/papers/324.pdf 1

[5] J. Woetzel and R. Koch, "Multi-camera real-time depth estimation with discontinuity handling on PC graphics hardware," in *ICPR*, 2004, pp. I: 741–744. [Online]. Available: http://dx.doi.org/10.1109/ICPR.2004.1334296 1

[6] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan, "Gpu implementation of belief propagation using cuda for cloud tracking and reconstruction," in *IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS 2008)*, 2008, pp. 1–4. 1

[7] V. Vineet and P. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1–8. 1

[8] L. Wang, M. Liao, M. L. Gong, R. G. Yang, and D. Nister, "High-quality real-time stereo using adaptive cost aggregation and dynamic programming," in *Proc. International Symposium on 3D Data Processing, Visualization and Transmission*, 2006, pp. 798–805. [Online]. Available: http://dx.doi.org/10.1109/3DPVT.2006.75 1

[9] M. L. Gong and Y. H. Yang, "Real-time stereo matching using orthogonal reliability-based dynamic programming," *IEEE Trans. Image Processing*, vol. 16, no. 3, pp. 879–884, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1109/TIP.2006.891344 1

[10] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *International Conference on Field Programmable Logic and Applications, 2009. FPL 2009*, 2009, pp. 126 – 131. 1

[11] G. L. Gimel'farb, "Probabilistic regularisation and symmetry in binocular dynamic programming stereo," *Pattern Recognition Letters*, vol. 23, no. 4, pp. 431–442, 2002. 2

[12] P. F. Fetzenszwald and D. P. Huttenlocher, "Efficient belief propagation for early vision," in *Proc. IEEE CS Conference on Computer Vision and Pattern Recognition*, vol. 1. IEEE CS Press: Los Alamitos, June 2004, pp. 261 – 268. 2

[13] V. Kolmogorov and R. Zabih, "Computing visual correspondence with occlusions via graph cuts," in *ICCV 2001*, July 2001. 2

[14] K. Jawed, J. Morris, T. Khan, and G. Gimel'farb, "Real time rectification for stereo correspondence," in *7th IEEE/IFIP Intl Conf on Embedded and Ubiquitous Computing (EUC-09)*, J. Xue and J. Ma, Eds. IEEE CS Press, 2009, pp. 277–284. 3

[15] N. Corp, "Cuda programming guide," 2009. [Online]. Available: http://www.nvidia.com/object/cuda_develop.html 3

[16] R. Kalarot and J. Morris, "Implementation of symmetric dynamic programming stereo matching algorithm using cuda," in *Sixteenth Korea-Japan Joint Workshop on Frontiers of Computer Vision*, 2010. 4, 5

[17] V. Gerdes, *Modular Rendering Tools*. www-student.informatik.uni-bonn.de/~gerdes/MRTStereo/index.html, 2001. 5