Accelerating Cost Aggregation for Real-Time Stereo Matching

Jianbin Fang*, Ana Lucia Varbanescu *, Jie Shen *,
Henk Sips *, Gorkem Saygili † and Laurens van der Maaten †
*Parallel and Distributed Systems Group
†Pattern Recognition and Bioinformatics Group
Delft University of Technology, Delft, the Netherlands
Email: {j.fang, a.l.varbanescu, j.shen, h.j.sips, g.saygili, l.j.p.vandermaaten}@tudelft.nl

***Abstract*—Real-time stereo matching, which is important in many applications like self-driving cars and 3-D scene reconstruction, requires large computation capability and high memory bandwidth. The most time-consuming part of stereo-matching algorithms is the aggregation of information (i.e. costs) over local image regions. In this paper, we present a generic representation and suitable implementations for three commonly used cost aggregators on many-core processors. We perform typical optimizations on the kernels, which leads to significant performance improvement (up to two orders of magnitude). Finally, we present a performance model for the three aggregators to predict the aggregation speed for a given pair of input images on a given architecture. Experimental results validate our model with an acceptable error margin (an average of 10.4%). We conclude that GPU-like many-cores are excellent platforms for accelerating stereo matching.**

***Index Terms*—Stereo Matching, Cost Aggregation, Performance Optimization, Performance Modeling, OpenCL, GPUs.**

I. Introduction

Stereo-matching algorithms aim to find an estimate of the depth inside a scene based on rectified stereo pairs of images [1]. The input to the stereo matching consists of two images of the same scene, a *reference* image and a *target* image, with each image displaying the scene from a different perspective along the x-axis. The goal is to accurately retrieve the *disparity*, i.e., the relative depth information for each pixel along the x-axis. The results are often used to estimate distance to an object in the scene, as objects corresponding to pixels of greater disparity are closer to the camera(s) than objects corresponding to pixels of lesser disparity.

Stereo algorithms can be divided into two main categories: (1) local algorithms and (2) global algorithms. Local algorithms [2]–[5] use aggregation of similarity measures around a local support region, i.e., the energy minimization is performed over local patches. By contrast, global algorithms [6]–[8] incorporate smoothness constraints on the depth estimates, leading to an energy minimization problem that involves all pixels in the images. In general, global algorithms outperform local algorithms in terms of accuracy due to the smoothness constraints, especially in non-textured image regions. However, global algorithms need to solve minimization problems that are generally NP-hard [9], prompting the use of approximations based on variational methods, graph cuts, or Monte Carlo simulation. Unfortunately, these approximations are still too slow for use in real-time algorithms, so local algorithms are preferred in this case.

Most consumer cameras operate at rates of 25 or 30 fps (frames per second), which may be sufficient for computer graphics due to human perception. However, many applications have even higher requirements, e.g., stereo algorithms used in automatically driving cars have to operate at a minimum of 100 fps [10]. Otherwise, in a scene where objects move at several dozen meters per second, those cameras will produce severe motion blur or temporal aliasing effects [10]. The high fps rates pose high demands for both computational capability and memory bandwidth. For this, an alternative for algorithm innovation is to use accelerators, e.g., GPUs, to speedup stereo matching without losing accuracy.

Born for computer graphics, GPUs are receiving a lot of attention in general-purpose computing due to their large performance potential. Currently, CUDA [11] and OpenCL [12] are the prevailing programming models on GPUs. CUDA is limited to NVIDIA GPUs, while OpenCL is an open standard for computing and it allows parallel programs to be executed on various platforms, addressing the issue of portability [13]. In this paper, we study the performance of local stereo-matching algorithms when implemented in OpenCL and executed on GPUs.

A. Motivation

Local stereo-matching algorithms comprise four main steps: (1) matching cost computation (CC), (2) cost aggregation (CA), (3) disparity estimation (DE), and (4) disparity refinement (DR) [1]. The matching cost computations (CC) involve calculating pixel-wise differences– based on the image intensities or gradients– between the two images in the stereo pair. These costs are aggregated over a support region of interest in the cost aggregation step (CA). In the disparity estimation step (DE), the disparity that minimizes the aggregated cost at a location is selected as the disparity estimate for that location (generally, using a winner-takes-all/WTA procedure). The disparity refinement step (DR) fine-tunes the estimated disparities using a simple type of smoothing. Figure 1 shows the amount of computation time spent in each of the aforementioned four components for three different CA solvers.

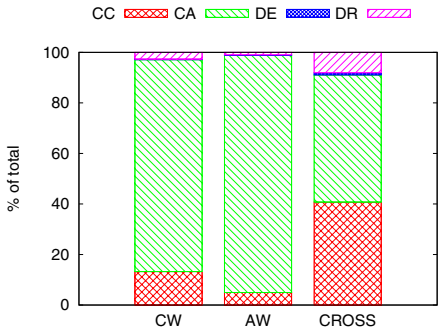


Fig. 1. The percentage of each component for three CA solvers (CW, AW, CROSS). The window radius for CW and AW is 10, and L, Tao for CROSS are 17, 20, respectively. We use (AD+Census) cost similarity measures to compute match cost (CC) [14], WTA (DE), and several optimizers in the refinement step (DR).

The CA step accounts for the most of the computation time in stereo matching (83.88%, 93.96%, and 50.28%, respectively). Therefore, we focus on accelerating CA in this paper.

Our main contributions are as follows:

- We analyze three commonly used state-of-the-art cost aggregators, and propose a unified representation that facilitates unified optimizations, performance prediction, and derivation of new algorithms (see Section II).
- We provide implementations for GPUs in OpenCL, perform multiple incremental optimizations for the three aggregators, and show that they are up to hundreds of times faster (see Section III).
- We present a performance model to predict the aggregation speed (in fps) of each aggregator, given a pair of images and a target machine (see Section IV).

Based on our empirical results, we conclude that GPU-like many-core processors are excellent platforms to satisfy the requirements of real-time (local) stereo matching.

II. Algorithms and the Representation

A. Aggregation Strategies

An aggregation strategy combines the image-matching costs over a (local) support region in order to obtain a reliable estimate of the costs of assigning a disparity d to image location (x, y). We investigate three commonly used local aggregation strategies: (1) constant window aggregation, (2) adaptive weight aggregation, and (3) cross-based aggregation.

Constant Window Aggregation (CW) is a straightforward aggregation of any similarity measure cost C over a constant-size neighborhood:

$$C_{CW}(x, y, d) = \sum_{\forall (x', y') \in \mathcal{N}(x,y)} C(x', y', d), \qquad (1)$$

where $\mathcal{N}(x, y)$ represents the set of pixels that are neighbors of pixel (x, y). CW assumes that every pixel in the window should share the same disparity. This assumption is violated in image regions in which the disparity is discontinuous (i.e. at object boundaries).

Adaptive Weight Aggregation (AW) [2] uses color similarity and proximity based weights as aggregation coefficients for the similarity measures pixels around pixels of interest. The dissimilarity between pixels $p = (x, y)$ in the reference image and $p_d = (x - d, y)$ in the target image is:

$$C_{AW}(p, p_d) = \frac{\sum_{q \in N(p), q_d \in N(p_d)} w(p, q) w'(p_d, q_d) C(p, d)}{\sum_{q \in N(p), q_d \in N(p_d)} w(p, q) w'(p_d, q_d)}, \qquad (2)$$

where p_d and q_d are the corresponding pixels in the target image of p and q with disparity shift of d. The weights $w(p, q)$ depend on the color similarity and Euclidean distance between pixels p and q:

$$w(p, q) = \exp\left(-\left(\frac{\|I(p) - I(q)\|^2}{\gamma_c} + \frac{\|p - q\|^2}{\gamma_s}\right)\right), \qquad (3)$$

where γ_c and γ_s are constant parameters that are set empirically. Calculation of target image weights $w'(p_d, q_d)$ is the same for target image pixels p_d and q_d.

Cross-Based Aggregation (CROSS) overcomes the problem of CW by constructing variable support for aggregation that depends on color similarity [4]. The first step of the algorithm is to construct a cross for each pixel inside the image. Four parameters are stored for each pixel, h_p^-, h_p^+, v_p^-, v_p^+, which identify the lengths of the four arms of the cross as shown for pixel p in Figure 2(a) (the light-shaded area). The decision on whether or not a pixel p' is included is made on its color similarity with pixel p as given in Equation 4.

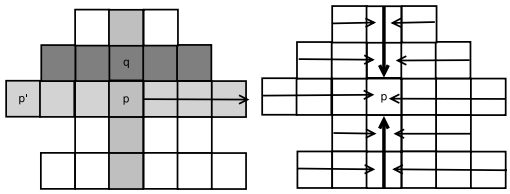


Fig. 2. Cross-based aggregation: (a) Cross Construction, (b) Horizontal Aggregation, and Vertical Aggregation.

$$\delta((x, y), (x', y')) = \begin{cases} 1, & \text{iff } |I(x, y) - I(x', y')| \leq \tau \\ 0, & \text{otherwise.} \end{cases} \qquad (4)$$

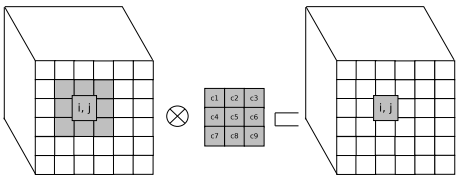
After calculating the arm lengths in four directions for each pixel, the final aggregation support region is constructed by integrating the horizontal arms of each neighboring pixel in vertical direction (e.g., the horizontal arms of pixel q in dark gray shown in Figure 2(a)). The same steps are also performed for the pixels of the target image. At the aggregation step, the intersection of the reference and the target support regions is aggregated. Orthogonal Integral Images (OIIs) can be used to speed up aggregation [4]. The technique separates the 2D aggregation procedure into two 1D aggregation steps: horizontal and vertical aggregation (see Figure 2(b)).

B. A Template for Cost Aggregation Kernels

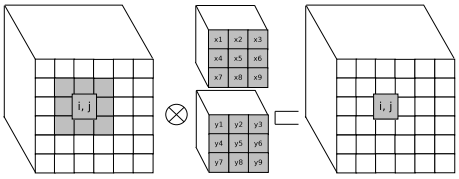
In order to give an overview of the three cost aggregation strategies, Figure 3 presents a kernel template for all of them. We see that the basic operation is a convolution between the input cost volume and the filter in the area determined by offsets(*offset_t, offset_b, offset_l, and offset_r*). We calculate the cost value (*ref_cost[]*, and *tar_cost[]*) at each *Disparity(D)* level for each pixel(x, y). When the *filter* and *offsets* are specified, the kernel template evolves into three different cost aggregation kernels (shown in Figure 4).

```
1  __kernel void aggregate(cost[], filter[], out_ref[], out_tar[]){
2
3    for each y in H
4      for each x in W
5        for each d in D{
6          res = 0;
7          for each y_ in <offset_t ... offset_b>{
8            for each x_ in <offset_l ... offset_r>{
9              res += cost[] OP filter[];
10           }
11             }
12         out_ref[] = res;
13         out_tar[] = res;
14       }
15 }
```

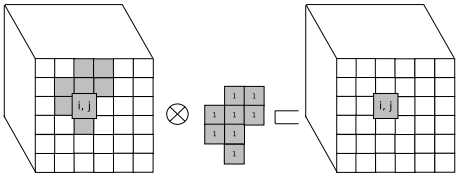
Fig. 3. A unified kernel template for cost aggregation.



(a) CW Aggregation



(b) AW Aggregation



(c) CROSS Aggregation

Fig. 4. Three cost aggregators for pixel(i, j) and disparity 0. As for AW, each pixel has a unique filter, thus forming a filter cube shown in (b).

- **Constant Window Aggregation (CW)**: as shown in Figure 4(a), the filter is filled with constant values, and the offset is constant for all the pixels (i.e., the window size is the same for all the pixels in the image).
- **Adaptive Weight Aggregation (AW)**: as illustrated in Section II-A, each pixel has its own filter, which is calculated according to the input images and is different between pixels. However, the filter size is fixed for all the pixels (Figure 4(b)). Further, two filters from the reference and target images are required for each pixel.
- **Cross-Based Aggregation (CROSS)**: as shown in Figure 4(c), there is no fixed filter for CROSS, but, for the sake of this common representation, we can define it as an irregular region filled with all ones. The offsets are calculated according to each pixel.

Using this template, we can develop new algorithms for cost aggregation. For example, the combination of AW and CROSS leads to a new cost aggregation solver with adaptive sized and weighted filters. Further, our template enables common optimizations at both architecture-level (e.g., exploiting data sharing for CW and AW) and algorithm-level (e.g., using the OII technique when filters are all filled with ones).

III. Implementations and Optimizations

We discuss here implementations and optimizations of the three cost aggregation solvers (CW, AW, and CROSS) on GPUs, and evaluate their overall performance [1].

A. OpenCL Implementations

Figure 5 shows our framework for implementing the local stereo matching algorithm in OpenCL [2]. Before porting kernels to the computing device, OpenCL has to set up a context and allocate the required device memory. Thereafter, we upload images (or video data) to the device (H2D), and compute disparity in four steps (CC, CA, DE and DR), as mentioned in Section I-A. We then transfer the disparity image back to the host after these four steps (D2H). When finalizing the program, these contexts and device memory space are released. As stated in Section I-A, we focus on the cost aggregations (CA, shaded in Figure 5).

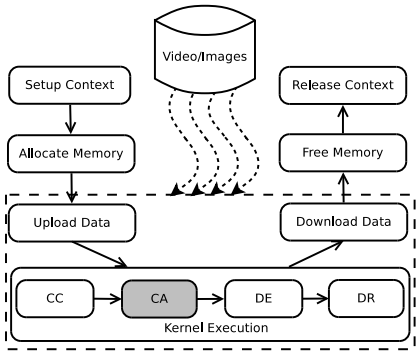


Fig. 5. Stereo matching framework implemented in OpenCL.

According to the kernel template for cost aggregations in Figure 3, we let each work-item process one pixel(x, y). Thus,

[1]The experiment setup is given in Section III-C.
[2]For OpenCL details, readers are referred to [12].

TABLE I
PERFORMANCE COMPARISON OF SIX WAYS TO MAP 2D WORK-ITEMS TO 3D DATA (MS).

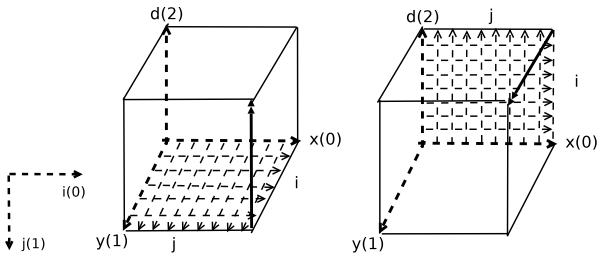
	$(i,j) \rightarrow (x,y)$	$(i,j) \rightarrow (x,d)$	$(i,j) \rightarrow (y,x)$	$(i,j) \rightarrow (y,d)$	$(i,j) \rightarrow (d,x)$	$(i,j) \rightarrow (d,y)$
512x512	0.42	0.50	1.06	6.33	2.58	7.08
1024x1024	1.50	1.72	4.19	24.48	12.15	28.70
2048x2048	5.80	7.01	16.76	93.92	64.00	117.31

a grid of $W \times H$ work-items is created (with an exception to be illustrated in Section III-B5).

B. Optimization Steps for CA on GPUs

We focus on five incremental optimization steps for cost aggregations: (1) mapping work-items to data, (2) exploiting data sharing, (3) caching data for coalescing, (4) unrolling loops, and (5) increasing parallelism.

1) Mapping Work-items to Data: The input into stereo cost aggregation is a 3-dimensional data matrix (the input cost volume), and a 2-dimensional thread grid is commonly used for GPU implementations. Then the question is how to efficiently map the work-items to the data matrix (i.e., the iteration space)? To maximize memory bandwidth on GPUs, neighboring work-items prefer accessing the spatially close data elements in groups, i.e., coalesced memory access [15]. For cost aggregation, we have $A_3^2 = 3 \times 2 = 6$ ways to map the work-item space to the data space. Figures 6(b) and 6(c) show two ways for the work-items to iterate through the data space, i.e., to force the work-items in the first dimension of a work-group to access the data elements in the first dimension of the input cost volume.

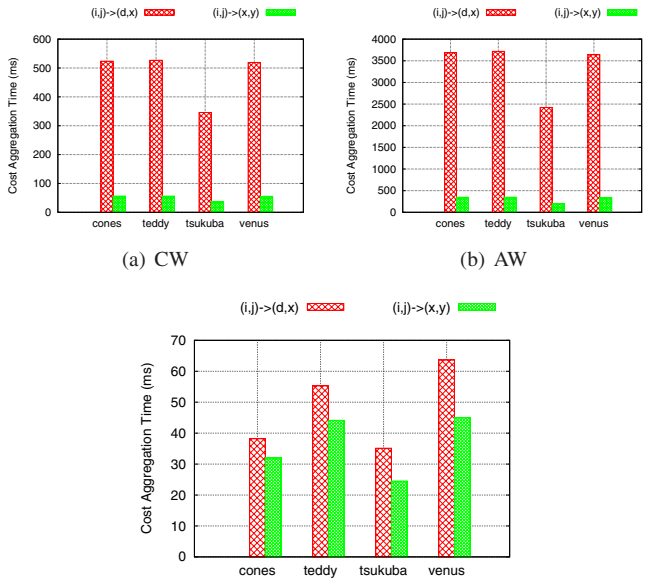


(a) (b) $(i,j) \rightarrow (x,y)$ (c) $(i,j) \rightarrow (x,d)$

Fig. 6. Iteration space. (i, j) are the 2 dimensions of the work-items space, and (x, y, d) are the 3 dimensions of the input cost volume.

We design micro-benchmarks for the six ways (of mapping 2D work-items to 3D data), and show how they perform on three data sets: 512×512, 1024×1024, 2048×2048 (when $D = 16$) in Table I. We see that the $(i,j) \rightarrow (x,y)$ and $(i,j) \rightarrow (x,d)$ mappings (shown in Figure 6(b) and 6(c)) perform much better than the other four (about 20x speedup over the case $(i,j) \rightarrow (d,y)$). Further, $(i,j) \rightarrow (x,y)$, which is our final preference (except the case mentioned in III-B5), performs slightly better than $(i,j) \rightarrow (x,d)$, due to more work-groups to hide latency.

We implement two mappings: $(i,j) \rightarrow (x,y)$ (our preference) and $(i,j) \rightarrow (d,x)$ (one of the 'bad' mappings) in the three cost aggregation kernels, and compare their performance (shown in Figure 7). We see that $(i,j) \rightarrow (x,y)$ shows a significant performance gain over $(i,j) \rightarrow (d,x)$ (the average speedups for CW and AW are 9.4x and 11.1x, respectively). Further, the CROSS kernel can be only accelerated by around 24%. This is because the filter for each pixel is adaptive that neighboring work-items may access data elements far away from each other (depending on the input images). In other words, coalesced access is not fully ensured for the CROSS kernel, which will be further discussed in Section III-B3.



(a) CW (b) AW

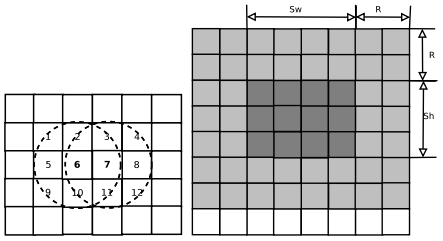
(c) CROSS

Fig. 7. Aggregation time of three aggregation strategies (CW, AW, and CROSS) on four datasets (cones, teddy, tsukuba, and venus) in Middlebury using two mappings ($(i,j) \rightarrow (x,y)$ and $(i,j) \rightarrow (d,x)$). The window radius for CW and AW is 10, and L, Tao for CROSS are 17, 20, respectively.

2) Exploiting Data Sharing: For the window-based aggregation (CW and AW), each work-item requires a block of data elements, and neighboring work-items share the overlapped data to aggregate cost. For example, when the window radius is 1 (shown in Figure 8), work-item 6 requires to load data elements at $\{1,2,3,5,6,7,9,10,11\}$, while work-item 7 needs data elements at $\{2,3,4,6,7,8,10,11,12\}$. Thus, we can use the on-chip *local memory* to share the data elements at $\{2,3,6,7,10,11\}$. In general, suppose the window radius is R (shown in Figure 8(b)), and the work-group size is $S_w \times S_h$, we calculate the *Shared Data Ratio* (SR) for a work-group using Equation 5. For the example shown in Figure 8(a), $SR = 2/3$. In theory, we expect to achieve $1/SR$ speedup for memory-bound applications, based on which our performance

model is built for kernels using local memory in Section IV.

$$SR = \frac{(S_w + 2 \times R) \times (S_h + 2 \times R)}{S_w \times S_h \times (2 \times R + 1)^2} \quad (5)$$



(a) An example. (b) A generic model.

Fig. 8. Data sharing demonstration.

The difficulty of using local memory lies in dealing with *halo data* [16], the data elements shaded in light-gray in Figure 8(b). As stated in the CUDA SDK, we use an extended work-group: the boundary work-items only participate in loading data from global memory to local memory, and remain idle for the other time. Figure 9 shows how the optimized CW performs when varying *R*. When the window is very small, we see a performance gain. However, when the *R* is larger than 5, the performance gain disappears because more and more work-items stay idle after loading data. Further, this approach is not scalable with the window size, i.e., as the radius of the window increases, the percentage of idle work-items increases and the size of a work-group may exceed its maximum limit [12]. Given the expected poor performance for AW, we choose not to implement AW with this approach.

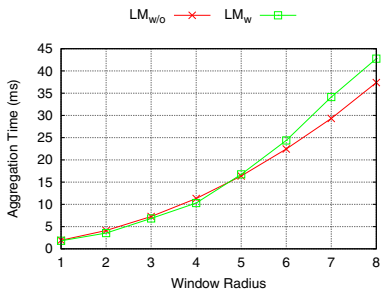
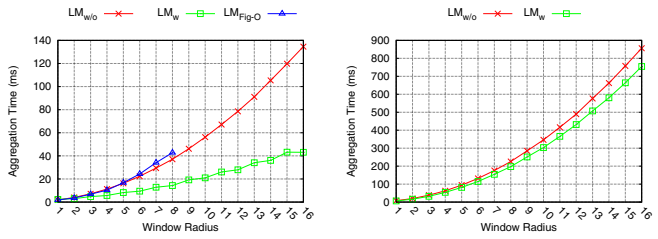


Fig. 9. Performance comparison of CW when using local memory for different window-radius. $LM_{w/o}$ represents the aggregation time without using local memory, and LM_w represents the case with that.

To tackle the lack of scalability, an alternative approach is to keep the geometry and the size of a work-group constant and load multiple data elements per work-item (the exact number of data elements processed by one work-item depends on the filter radius and the work-group size). From Figure 10, we see the performance improvement using this revised approach. The average speedup is 2.3 for CW and 1.2 for AW, respectively. Using local memory can accelerate AW less significantly, because AW needs to load two other elements (i.e., filter elements) from the global memory, which becomes the performance bottleneck.



(a) CW (b) AW

Fig. 10. Performance comparison when using local memory for two aggregation methods (CW, AW) and different window-radius. $LM_{w/o}$ represents the aggregation time without using local memory, and LM_w represents the case with that. LM_{Fig-O} means that we use local memory in the way shown in Figure 9.

3) Caching Data for Coalesced Access: We see from Figure 4(c) that the CROSS cost aggregation requires an adaptive area of data elements for each pixel. Thus, neighboring work-items will access the global memory possibly in an uncoalesced way. Local memory can be used to explicitly convert a scattered access pattern to a regular (coalesced) pattern for global memory access [15]. To be specific, when using local memory, the coalesced access from global memory to local memory is ensured and operating the local memory itself has no coalescing constraints. Thus, for CROSS, we first load data elements within the area of radius L (the predefined maximum limit on radius) from global memory in a coalesced way, and then use the required data elements from local memory.

In Figure 11, we see that using local memory for caching can improve the CROSS performance by 1.5x on average. Although we load more data elements than what we need in this situation, we have achieved better performance, due to the guarantee of coalesced data access from global memory.

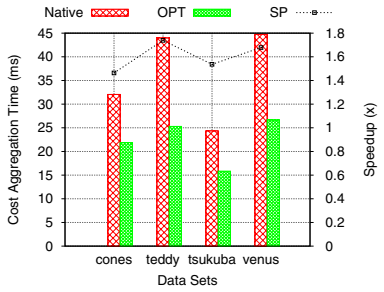


Fig. 11. CROSS performance on four data sets when caching data wiht local memory ($L = 17$, $Tao = 20$). OPT represents the kernel with caching.

4) Unrolling Loops: Loop unrolling is an optimization technique in which the body of a suitable loop is replaced with multiple copies of itself, and the control logic of the loop is updated at well [17]. In this way, we can reduce the number of branch instructions, thus improving performance. For-loops exist in cost aggregation kernels due to the heavy usage of filters (shown in Figure 12). Thus, we generate the kernels in the fully loop-unrolled fashion and then evaluate their performance.

Figure 13(a) shows that we can further improve the CW

```
1  // ... head code ...
2  // cache[] represents the usage of local memory
3  // l_c and l_r is the local column and row index
4  if(l_c<16 && l_r<16){
5      float cost_aggr=0;
6      for(int y_ = 0; y_ < wnd_side; y_++){
7          for(int x_ = 0; x_ < wnd_side; x_++){
8              int x_start = l_c + x_;
9              int y_start = l_r + y_;
10             cost_aggr += cache[y_start*length+x_start]*filter[];
11         }
12     }
13 }
14 // ... tail code ...
```

Fig. 12. Cost aggregation kernels with loops.

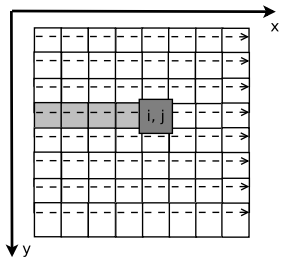
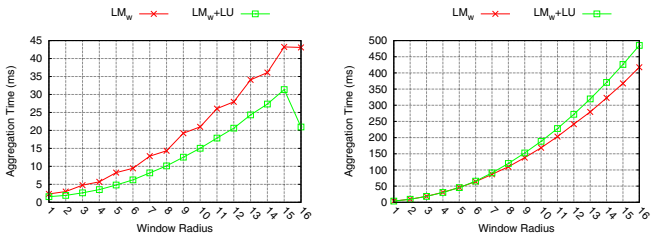


Fig. 14. Data dependency when calculating prefix sum in x direction: to calculate the cost value at (i, j), we use Equation $cost(i, j) = \sum_{i' \le i, j'=j} cost(i', j')$. So the integrated cost value at (i, j) depends on all its previous elements, namely the elements shaded in gray.

performance using local memory plus loop unrolling (the average speedup 1.5x). Additionally, we see that using loop unrolling on AW decreases the performance (in Figure 13(b)). This slow-down is due to register pressure (i.e., each work-item of AW needs to load two more filter elements from global memory for each iteration). Thus, we choose not to perform loop unrolling on AW. As for CROSS, the irregularity of filters hinders the loop-unrolling during compiling time.



(a) CW (b) AW

Fig. 13. Performance comparison between with and without loop unrolling for CW, AW and different window-radius. LM_w represents the aggregation time only using local memory (no loop unrolling), and $LM_w + LU$ represents the case with both local memory and loop unrolling.

5) Increasing Parallelism: When we parallelize aggregation kernels, letting each work-item process one pixel is a natural choice. However, it becomes difficult to do so when using *OII* to pre-compute prefix sum for the CROSS cost aggregation (see Section II-A and [4] on *OII*), because of the data dependency in the x or y direction (e.g., data dependency in the x direction shown in Figure 14).

In this case, we maximize parallelism by using a third dimension, namely D (representing *Disparity*). Specifically, we map the 2D work-items to the 2D elements in the (x, d) or (y, d) plane, and iterate over the third dimension (y or x, respectively). Once the third dimension is used, more work-groups can fully populate the compute unit to hide latency, leading to high occupancy [15]. We show that the performance differs significantly between using 1D and 2D parallelism in Table II. The speedup ranges from 1.4x to 4.1x, depending on the size of the images. For the larger images, the gain is less significant. In fact, for very large images, we expect that the performance gain using 2D parallelism will disappear, as there will be enough work-groups to hide latency in one dimension. However, in real-time stereo matching, the commonly used images, e.g., the Middlebury dataset [18], are smaller than or close to 512×512, and they will benefit a lot from this 2D parallelism model.

TABLE II
PREFIX SUM EXECUTION TIME USING 1D AND 2D PARALLELISM (D=16).

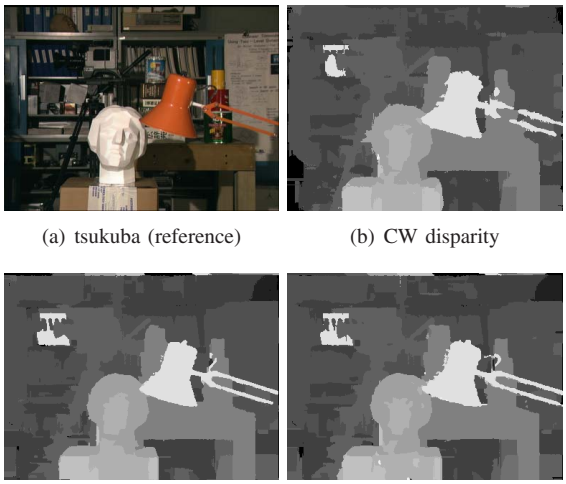
	512x512	1024x1024	2048x2048	4096x4096
1D(ms)	24.03	49.28	122.73	421.58
2D(ms)	5.81	22.18	76.83	299.11
speedup(x)	4.14	2.22	1.60	1.41

C. Overall Performance

All the experiments (except those in Section III-C3) are performed on a NVIDIA Quadro5000 Fermi GPU, connected to the host (Intel Xeon X5650). The card has Compute Capability 2.0 and consists of 352 cores divided among 11 multiprocessors. The amount of local memory available per multiprocessor is 48K. We compile all the program with the OpenCL implementation from CUDA version 4.2 and GCC version 4.4.3. We use four image pairs from the Middlebury dataset: cones, teddy, tsukuba, and venus. We perform initial experimental analysis on accuracy (in Section III-C1), and focus on speed (in Section III-C2 and III-C3). For the speed part, we use the average results of these four data sets.

1) Accuracy: Figure 15 shows the disparity results for tsukuba. Compared with the ground truth image, the error rates are 8.29%, 4.28% and 5.41% for CW, AW and CROSS, respectively, i.e., $AW > CROSS > CW$ in terms of accuracy. As accuracy analysis or improvement is not the focus of this paper (we focus on improving the speed of the three aggregation solvers without a loss in accuracy), we will not dive into more details.

2) Speed on the Quadro5000: We show the aggregation time ($Sequential$ versus $Optimized$) for CW, AW, and CROSS in Figure 16. From Figure 16(a), we see that AW is the most time-consuming aggregation solver (although it can achieve the most accurate disparity map shown in Figure 15), and its aggregation time increases exponentially with the window-radius. When using very small windows, CW runs



(a) tsukuba (reference) (b) CW disparity

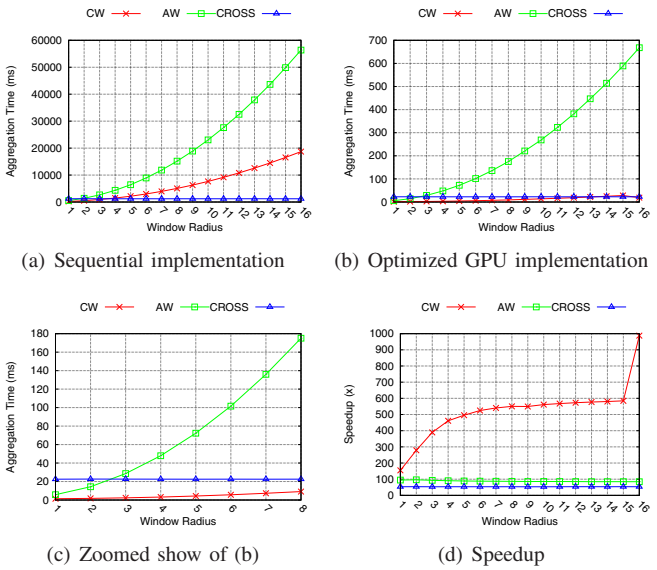
(c) AW disparity (d) CROSS disparity

Fig. 15. Accuracy comparison: (a) the tsukuba data set; (b)-(c) are the output disparity images for CW, AW, and CROSS using AD+Census cost similarity measures [14], WTA, and several optimizers in the refinement step.

faster than CROSS. However, as the window becomes bigger, the time for CW increases, while it remains stable for CROSS (that is because CROSS is window-independent).

As shown in Figures 16(b), 16(c) and 16(d), the optimization steps improve the aggregation performance significantly. The speedups stay around 90 for AW, while we can achieve more performance improvement for CW (the average speedup is 523) with the increasing of window sizes. This is due to more data sharing within a work-group. Further, we can achieve relatively small speedup for the CROSS solver (53x on average), due to loading extra data elements to local memory.

As can be seen from Figure 16(d), when the window radius is 16, the performance improves dramatically (compared with



(a) Sequential implementation (b) Optimized GPU implementation

(c) Zoomed show of (b) (d) Speedup

Fig. 16. Performance comparison of sequential implementation and the optimized GPU implementation.

the case when the window radius is 15), because all work-items in a work-group are active when loading data from global memory to local memory. Further, we can verify that the achieved GFLOPs is below the theoretical GFLOPs: the size of tsukuba is 384×288, the filter radius is 16, $D = 16$, and it takes 13.37 ms to perform the CW aggregation. The achieved computational throughput is 144 $GFLOPs$ (see Equation 6), which is lower than the theoretical peak of Quadro5000 - 359 $GFLOPs$ for addition only operations.

$$\frac{384 \times 288 \times 16 \times 33 \times 33}{13.37} \times 10^{-3} = 144\ GFLOPs \quad (6)$$

3) Speed on the Low-end GPU: We evaluate our parallel implementation on a low-end GPU: NVIDIA Quadro NV 140M (2 multiprocessors, 16 cores each). Figure 17 shows the speedup of CW compared with its sequential implementation. We see that the optimized CW can obtain impressive speed-up factors, ranging from 5x to around 60x. One the other hand, the AW fails to run, because it consumes a lot of device memory, which exceeds the maximum limits. As for CROSS, it suffers a 25% performance loss, because the older generation NV 140M imposes more severe constraints on effective memory accesses.

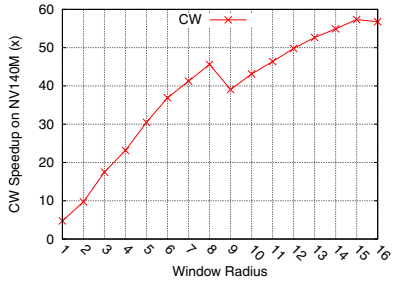


Fig. 17. Speedup of the optimized CW over its 1-thread implementation on the low-end NV140M.

4) Putting it all together: As mentioned in Section III-A, the local stereo-matching algorithm consists of six steps when implementing it in OpenCL (apart from the context management). After performing all these optimization steps, the percentages of each component are shown in Figure 18. We see that CC becomes the performance bottleneck, rather than CA (although we perform the optimization on the other steps besides CA). Thus, we will study the local stereo-matching algorithm as a whole to get better performance for further work.

To summarize, we have identified and performed five optimization steps on the aggregation solvers (CW, AW, and CROSS). Compared with the sequential implementations, the experimental results show significant speedup on NVIDIA Quadro5000: CW is around 500 times faster (50-700 fps), while AW and CROSS are tens of times faster (AW reaches 1.5-170 fps and CROSS 45 fps). Thus, meeting real-time requirements is feasible. Further, based on the standard template plus these key optimizations steps, a straightforward extension of this work is to build an auto-tuner for stereo matching.

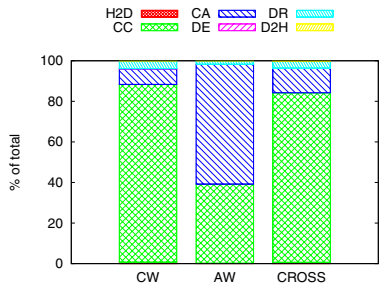


Fig. 18. The percentage for each component using OpenCL. The window radius for CW and AW is 10, and L, Tao for CROSS are 17, 20, respectively. We use (AD+Census) cost similarity measures to compute match cost (CC) [14], WTA (DE), and several optimizers in the refinement step (DR).

IV. Performance Modeling and Prediction

In this section, we build a performance model to predict the cost aggregation speed of the three kernels. This model can (1) provide users with intuitive answers on whether a given machine and an aggregation solver can meet their fps requirements, and (2) give programmers an upper bound target for (further) performance optimizations. These cost aggregation kernels are memory-bound on most state-of-the-art architectures (the $arithmetic\ intensity$ for CW and AW is 1/4 and 1/12, respectively) [19]. Based on this, we build our model as follows.

A. Theoretical Model

We calculate the data requirements *M* for CW, AW, and CROSS using Equation 7.

$$M = \begin{cases} W \times H \times D \times (2 \times R + 1)^2 \times size(T) & [CW] \\ W \times H \times D \times (2 \times R + 1)^2 \times size(T) \times 3 & [AW] \\ W \times H \times D \times (2 \times H_{AVG}) \times size(T) & [CROSS] \end{cases} \tag{7}$$

where (W, H, D) are the three dimensions of the input cost volume, R is the radius of filters, and T represents the data type. For AW, two more data elements are required from filters, thus $\times 3$. The CROSS aggregation differs in: (1) the window sizes vary on each pixel, and (2) CROSS uses OII to improve aggregation performance. Thus, we use H_{AVG} to represent the average window height in the vertical direction, and calculate it statistically from the input cost volume (e.g., the H_{AVG} for tsukuba is 17).

When using local memory, the number of data elements is reduced to SR' (using Equation 8) which is based on SR in Equation 5.

$$SR' = \frac{M_u + (S_w + 2 \times R) \times (S_h + 2 \times R)}{M_u + S_w \times S_h \times (2 \times R + 1)^2} \tag{8}$$

M_u represents the amount of data elements from global memory directly (i.e., no usage of local memory), while the other variables have the same meaning as in Equation 5.

The data amount M_l using local memory is given by:

$$M_l = M \times SR' \tag{9}$$

Assuming the theoretical bandwidth of the target machine is B, we calculate fps_{ideal} (the theoretical fps) without and with the usage of local memory in Equation 10.

$$fps_{ideal/wo} = B/M, fps_{ideal/wi} = B/M_l \tag{10}$$

B. Model Calibration

fps_{ideal} can be achieved only when fully utilizing the memory bandwidth, which is difficult in real-world applications. Thus, we introduce C^* to represent the $usage$ of memory bandwidth, and calibrate our model by:

$$fps \sim fps_{real} = C^* \times fps_{ideal} \tag{11}$$

As for the data set tsukuba, its dimension is 384×288. When performing cost aggregation on NVIDIA Quadro5000, the theoretical memory bandwidth is 120 GB/s. We use the model described by Equations 7-10 to calculate the theoretical fps (see Table III). Further, we run two sets of experiments aiming to measure the real fps: without and with local memory (see Table IV and Table V, respectively).

TABLE III
THEORETICAL FPS ($LM_{w/o}$ REPRESENTS THE FPS FOR THE KERNELS WITHOUT USING LOCAL MEMORY, WHILE LM_w REPRESENTS THE CASE WITH THAT; LOCAL MEMORY IS USED ONLY IN AW AND CW).

	CW		AW		CROSS	
R	$LM_{w/o}$	LM_w	$LM_{w/o}$	LM_w	$LM_{w/o}$	LM_w
1	2022.72	14383.76	674.24	944.92	535.42	-
2	728.18	11650.84	242.73	353.06	535.42	-
3	371.52	9628.80	123.84	182.24	535.42	-
4	224.75	8090.86	74.92	110.83	535.42	-
5	150.45	6893.99	50.15	74.41	535.42	-
6	107.72	5944.31	35.91	53.38	535.42	-
7	80.91	5178.15	26.97	40.14	535.42	-
8	62.99	4551.11	21.00	31.28	535.42	-
9	50.43	4031.43	16.81	25.06	535.42	-
10	41.28	3595.94	13.76	20.52	535.42	-
11	34.41	3227.38	11.47	17.12	535.42	-
12	29.13	2912.71	9.71	14.49	535.42	-
13	24.97	2641.91	8.32	12.43	535.42	-
14	21.65	2407.20	7.22	10.77	535.42	-
15	18.94	2202.43	6.31	9.43	535.42	-
16	16.72	2022.72	5.57	8.32	535.42	-

TABLE IV
MEASURED FPS WITHOUT USING LOCAL MEMORY IN KERNELS.

	CW		AW		CROSS	
R	fps	usage(%)	fps	usage(%)	pfs	usage(%)
1	776.35	38.38	227.37	33.72	41.69	7.79
2	369.84	50.79	85.87	35.38	41.61	7.77
3	211.87	57.03	44.78	36.16	42.15	7.87
4	135.80	60.42	27.54	36.76	41.55	7.76
5	94.10	62.55	18.62	37.13	42.00	7.84
6	69.03	64.09	13.36	37.20	41.60	7.77
7	52.72	65.16	10.00	37.08	41.54	7.76
8	41.60	66.04	7.85	37.41	41.51	7.75
9	33.64	66.70	6.32	37.59	41.38	7.73
10	27.73	67.18	5.13	37.29	41.41	7.73
11	23.27	67.63	4.28	37.35	41.97	7.84
12	19.79	67.96	3.61	37.19	40.98	7.65
13	17.06	68.30	3.09	37.14	40.66	7.59
14	14.85	68.59	2.70	37.49	41.99	7.84
15	13.04	68.81	2.36	37.44	41.66	7.78
16	11.54	69.03	2.08	37.26	41.54	7.76

As we can see from Tables IV and V, the $usage$ (the ratio of the measured fps to the theoretical fps) is much lower than

TABLE V
MEASURED FPS WHEN USING LOCAL MEMORY IN KERNELS (WE IGNORE CROSS IN THIS CASE DUE TO LACK OF INFORMATION ON SR).

	CW		AW		CROSS	
R	fps	usage(%)	fps	usage(%)	pfs	usage(%)
1	937.09	6.51	270.20	28.60	-	-
2	764.04	6.56	105.84	29.98	-	-
3	558.95	5.81	54.14	29.70	-	-
4	427.51	5.28	32.70	29.50	-	-
5	316.46	4.59	21.80	29.30	-	-
6	245.09	4.12	15.65	29.31	-	-
7	191.28	3.69	11.69	29.12	-	-
8	153.85	3.38	9.05	28.93	-	-
9	121.94	3.02	7.24	28.88	-	-
10	102.31	2.85	5.90	28.75	-	-
11	86.38	2.68	4.95	28.92	-	-
12	74.16	2.55	4.16	28.71	-	-
13	64.16	2.43	3.58	28.80	-	-
14	56.14	2.33	3.11	28.83	-	-
15	49.34	2.24	2.71	28.77	-	-
16	74.74	3.70	2.39	28.74	-	-

100%. Apart from the aforementioned incomplete bandwidth utilization, three more factors lead to this: (1) kernel branches, (2) the overhead of address calculation instructions, and (3) the overhead of using local memory. Thus, we take all these factors into account by approximating $C^* = usage(R)$, where $usage$ is empirically measured, and it is correlated with the window-radius R.

C. Model Validation

To validate our model, we use the three other datasets from Middlebury: cones (450×375), teddy (450×375), venus (434×383)[3]. Figure 19 shows the comparison of measured fps (fps_m) and predicted fps (fps_p) for CW and AW. It can be seen that we can correctly predict the results (the average error rate is 6.7% for CW and 16% for AW). As for CROSS (with adaptive window size for each pixel), the H_{AVG} is 11 for cones, and the average error rate is 6.6%.

When using local memory in kernels (for CW and AW only), we use the model to predict the aggregation speed, and show the results in Figure 20. The average error rate is 4.9% and 17.9% for CW and AW, respectively.

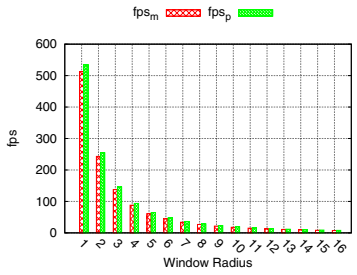
Overall, the results show that our performance model is able to predict the performance of stereo aggregation with acceptable accuracy. For CW and CROSS, the model only requires some further calibration. For AW, increasing the prediction accuracy might require a more detailed calibration (work in progress).

V. RELATED WORK

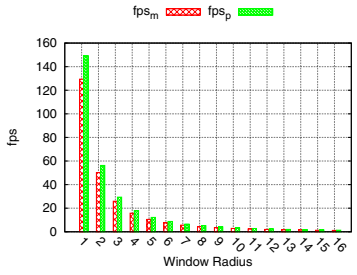
In this section, we present previous work on GPU-assisted stereo matching (we have mentioned related work on stereo cost aggregation solvers in Section II).

In [20], Wang et al. introduce an adaptive aggregation step in a dynamic-programming (DP) stereo framework, and utilize the vector processing capability and parallelism in commodity graphics hardware, increasing the performance by over two orders of magnitude. Their performance improvements are

[3]Due to space limits, we only present the results on *cones*. The other results can be provided upon request.

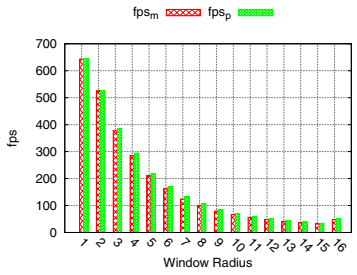


(a) CW

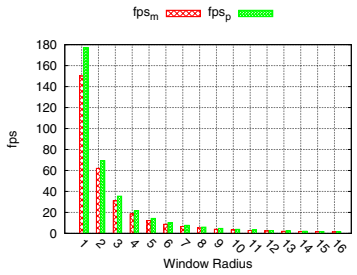


(b) AW

Fig. 19. Performance prediction without local memory. fps_m represents the measured fps by experiments, while fps_p is the predicted fps.



(a) CW



(b) AW

Fig. 20. Performance prediction with local memory. fps_m represents the measured fps by experiments on kernels using local memory, while fps_p is the predicted fps considering local memory.

mainly based on the usage of texture memory. Gong [21] gives an implementation of six aggregation approaches, two of which (CW and AW) are also implemented in our work, and evaluates them on a real-time stereo platform in terms of both accuracy and speed. The used platform includes programmable graphics hardware. Compared with our work, both these solutions use older GPU programming methods and focus on different optimizations.

More recently (since 2006), GPUs have evolved into computing devices solving general-purpose problems. At the same time, CUDA [11] and OpenCL [12] make programming GPUs easier. As a result, more research effort is put in using GPUs for stereo matching. In [22], the authors propose a GPU-based stereo system, consisting of hardware-aware algorithms and code optimization techniques, which improves both the accuracy and the speed of stereo-matching.

Scott Grauer-Gray et al. [23] explore methods to optimize a CUDA-implementation of belief propagation (i.e., a global stereo matching method). They focus their investigation on the optimization space of using local, shared, and register memory options for data storage on the GPU. Mei et al. [14] present a near real-time stereo system with accurate disparity results using multiple techniques, viz., AD-Census costs, cross-based support regions (which is also evaluated in our work), scanline optimization, and a systematic refinement process. The results are achieved on NVIDIA GTX480 within 0.1 seconds. By contrast, our work focuses larger sets of aggregators, and investigates the performance benefits of a unified representation and optimizations on them.

In this still sparse space of GPGPU-enabled stereo-matching, our work proposes a unified approach for multiple cost-aggregation kernels, in terms of both implementations and optimizations; furthermore, because of this unified approach, we are able to propose the first (to the best of our knowledge) simple and effective performance model to predict an upper bound for the GPU-based stereo-matching speed on a given platform.

VI. Conclusion

In order to meet real-time requirements for cost aggregations, we study three typical aggregation solvers on GPUs: CW, AW, and CROSS. For the three aggregators, we devise a unified representation and implementation, and we perform five incremental optimization steps: (1) mapping work-items to data, (2) exploiting data sharing, (3) caching data for coalescing, (4) unrolling loops, and (5) increasing parallelism. Experimental results show that we can significantly boost the performance of cost aggregations without a loss in accuracy. Further, we present a prediction model to estimate the fps that cost aggregation can achieve on a given platform. Using these estimates, a better fit between cost aggregators, real-time requirements, and hardware platforms can be achieved.

Since we have implemented the stereo software in OpenCL, we are currently working on the performance analysis on CPUs (i.e., we investigate how these aggregation solvers perform on multi-core architectures and their optimization space). Furthermore, additional validation tests and calibration experiments are needed before the final refinement and generalization of the prediction model for multi-cores.

Acknowledgment

We would like to thank Daniel Scharstern for making the Middlebury test datasets publicly available. Thank the Advanced School for Computing and Imaging (ASCI) for providing the experimental platforms (DAS-4). This work is partially funded by CSC (China Scholarship Council), and the National Natural Science Foundation of China under Grant No.61103014.

References

[1] D. Scharstein, R. Szeliski, and R. Zabih, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," in *IEEE SMBV*, 2001.

[2] K. Yoon and I. Kweon, "Locally adaptive support-weight approach for visual correspondence search," in *CVPR*, 2005.

[3] D. Min, J. Lu, and M. Do, "A revisit to cost aggregation in stereo matching: How far can we reduce its computational redundancy?," in *ICCV*, 2011.

[4] K. Zhang, J. Lu, and G. Lafruit, "Cross-based local stereo matching using orthogonal integral images," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 7, pp. 1073–1079, 2009.

[5] C. Richardt, D. Orr, I. Davies, A. Criminisi, and N. Dodgson, "Real-time spatiotemporal stereo matching using the dual-cross-bilateral grid," *ECCV*, pp. 510–523, 2010.

[6] P. Felzenszwalb and D. Huttenlocher, "Efficient belief propagation for early vision," *IJCV*, 2006.

[7] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 11, pp. 1222–1239, 2001.

[8] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, pp. 1124–1137, 2004.

[9] F. Barahona, "On the computational complexity of Ising spin glass models," *Journal of Physics A: Mathematical and General*, vol. 15, pp. 3241+, Jan. 1999.

[10] S. Meister, B. Jähne, and D. Kondermann, "An outdoor stereo camera system for the generation of Real-World benchmark datasets," *Optical Engineering*, 2011.

[11] NVIDIA Inc., "CUDA Toolkit 4.1." http://developer.nvidia.com/cuda-toolkit-41, Februay 2012.

[12] Khronos OpenCL Working Group, *The OpenCL Specification*, November 2011.

[13] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *ICPP*, 2011.

[14] X. Mei, X. Sun, M. Zhou, S. Jiao, H. Wang, and X. Zhang, "On building an accurate stereo matching system on graphics hardware," *GPUCV*, 2011.

[15] NVIDIA Inc., *NVIDIA OpenCL C Programming Guide*, June 2011.

[16] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: optimizing GPU memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), ACM, 2011.

[17] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–11, IEEE, Apr. 2010.

[18] D. Scharstein and R. Szeliski, "Middlebury Stereo Datasets." http://vision.middlebury.edu/stereo/, Februay 2012.

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[20] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister, "High-Quality Real-Time stereo using adaptive cost aggregation and dynamic programming," in *3DPVT*, 2006.

[21] M. Gong, R. Yang, L. Wang, and M. Gong, "A performance study on different cost aggregation approaches used in Real-Time stereo matching," *Int. J. Comput. Vision*, vol. 75, pp. 283–296, Nov. 2007.

[22] W. Yu, T. Chen, F. Franchetti, and J. C. Hoe, "High performance stereo vision designed for massively data parallel platforms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, pp. 1509–1519, Nov. 2010.

[23] S. G. Gray and J. Cavazos, "Optimizing and auto-tuning belief propagation on the GPU," in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, (Berlin, Heidelberg), pp. 121–135, Springer-Verlag, 2011.