High Performance Stereo Vision Designed for Massively Data Parallel Platforms

Wei Yu, and Tsuhan Chen, *Fellow, IEEE,* and Franz Franchetti, and James C. Hoe

***Abstract*—Real-time stereo vision is attractive in many applications like robot navigation and 3D scene reconstruction. Data parallel platforms, e.g. GPU, is often used for real time stereo, because many stereo algorithms involve a large portion of data parallel computations. In this paper, we propose a stereo system on GPU which pushes the Pareto-efficiency frontline in the accuracy and speed trade-off space. Our design is based on hardware-aware algorithm design approach. The system consists of new algorithms and code optimization. We emphasize on keeping the highly data parallel structure in algorithm design such that the algorithms can be effectively mapped to massively data parallel platforms. We propose two stereo algorithms named exponential step size adaptive weight (ESAW) and exponential step size message propagation (ESMP). ESAW reduces computational complexity without sacrificing disparity accuracy. ESMP is an extension of ESAW, which incorporates the smoothness term. ESMP offers additional choice in the accuracy and speed trade-off space. When mapping an algorithm to a hardware platform, there are many choices to be made to achieve the best performance. We discuss code optimization techniques widely applied in the performance tuning community, rather than optimizing the code in an 'ad hoc' manner. We compare our results with state-of-the-art real-time stereo vision systems. Experiment results demonstrate a speed-up factor of 2.7 to 8.5 over existing systems at comparable disparity accuracy.**

***Index Terms*—stereo, real-time, multi-core, data parallel, GPU, code optimization.**

I. Introduction

THE goal of stereo vision is to reconstruct a disparity map (reciprocal of depth) from two views. Both accuracy and speed are important metrics in designing real-time stereo systems. Existing stereo systems usually performs well in one aspect but not good in the other, because they focus on either improving accuracy or code optimization for an existing algorithm. We take a different approach by designing algorithms in aware of hardware features. Data parallel architectures are widely used for real time stereo, because for most stereo algorithms a large portion of the computing time is spent on data parallel processing. The hardware platform we use is GPU (Graphics Processing Unit), an instance of massively data parallel architectures. Our goal is to design stereo algorithms that can be effectively mapped to such platform.

Stereo accuracy can be evaluated by error rate, which is the average percent of bad pixels (the same as the last column "average percent of bad pixels" in Middlebury stereo evaluation online system [26]) of all four benchmark datasets (Tsukuba, Venus, Teddy, and Cones). Speed is measured by the system throughput, i.e. millions of disparity per second (MDS).

In terms of accuracy, state-of-the-art stereo algorithms can be categorized into 3 classes: very good quality (error rate below 7.0), good quality (error rate in between 7.0 and 11.0), and not good quality (error rate above 11.0). Stereo algorithms producing very good disparity quality usually involve complex computations for global optimization, segmentation, plane fitting and occlusion handling, etc. To our best knowledge, none of the algorithms in the first class (very good quality) have been implemented in a real-time system yet. The only near real-time solution we know of is proposed by Yang, Q., et al. in [16], achieving error rate of 5.8 at system throughput of 9.4 MDS. At this throughput, it takes 1.3s to process a stereo image pair of size 384×512 and 60 disparity levels.

A number of real-time systems for algorithms in the second class (good quality) have been proposed [4], [15], [11]. All of them have been implemented on graphics cards. The fastest among them is the system proposed by Gong et al. [4], achieving error rate of 11.0 at system throughput of 124 MDS. At this throughput, it takes 96ms to process a stereo image pair of size 384×512 and 60 disparity levels. Therefore, to improve system throughput at good disparity accuracy remains a challenging problem.

Contribution. The main contributions in this paper is a stereo system built on hardware-aware software design concept. We keep the highly data parallel structure in algorithm design, such that the algorithms can be efficiently mapped to a GPU platform. We propose two algorithms and code optimization.

- The two algorithms are exponential step size adaptive weight (ESAW) and exponential step size message propagation (ESMP). ESAW allows cost information from distant pixels to propagate to the center pixel within a few iterations. ESMP is an extension of ESAW by incorporating the smoothness term commonly used in belief propagation for global stereo. ESMP can improve the disparity accuracy at the cost of lower throughput.
- We discuss various choices in code optimization and analyze the trade-offs in efficiency. The methodologies are widely used in performance tuning community, but rarely found in the vision literature.

Organization. In Section II we present the necessary background and related work. First, we explain hardware platform features and code optimization guidelines. Then we discuss existing real-time or near real-time stereo systems. In Section III, we introduce two stereo algorithms and analyze their complexity and accuracy. In Section IV, we present code

W. Yu, F. Franchetti, and J. C. Hoe are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213 USA e-mail: wy@andrew.cmu.edu, franzf@ece.cmu.edu, jhoe@ece.cmu.edu

T. Chen is with the School of Electrical and Computer Engineering, Cornell University, tsuhan@ece.cornell.edu.

Manuscript received ; revised

optimization techniques to efficiently map both algorithms to the hardware. Section V presents experiment results and a comparison with existing systems. Finally, we offer conclusions in Section VI.

II. Background and Related Work

We provide background of the hardware platform and basic optimization guidelines. We also discuss existing work on designing real-time stereo systems.

A. Hardware platform

Stereo vision demonstrates intensive fine-grained data parallelism, which can take advantage of the massively data parallel architectures. GPU is an instance of such data parallel platforms. The GPU we used is NVIDIA GeForce GTX 8800, with CUDA (Computer Unified Device Architecture) programming interface.

GPU architecture features. The GTX 8800 is a hierarchical architecture consisting of a total of 128 cores organized into 16 stream multi-processors (SM), each SM containing 8 stream processors (SP), or cores. Each SP runs at 1.35GHz, and has one 32-bit single-precision floating point multiply-add arithmetic unit. Fully pipelined arithmetic units yield a total of $1.35\text{GHz} \times 16\text{SM} \times (8 \times 2)\text{flop/SM} = 345.6\text{Gflop/s}$ theoretical peak performance.

The memory system of the GTX 8800 comprises 768MB off-chip global memory, 64kB on-chip cache for texture memory, 16kB on-chip cache for constant memory per SM, 16kB shared memory per SM, 8k 32-bit registers and local memory for register spilling purpose. Off-chip memory access exhibits very long latency (200–300 cycles if L1 hit and 400-600 cycles if L1 miss); latency for on-chip texture cache is about 100 cycles; and accessing other on-chip memory is very fast (1–2 cycles). Though the GTX 8800 features a high off-chip bandwidth of 86.4GB/s, it is still easy to saturate the memory bandwidth given the high peak computing power.

CUDA GPU Programming model. The GTX 8800 supports single program multiple data (SPMD) programming model. The computation task is coded into *kernel* functions. Each *kernel* is executed by multiple threads concurrently on different data. Each *kernel* creates a single *grid* that consists of multiple *thread blocks*. Every *thread block* is assigned to execute on one SM. Each *thread block* is further partitioned into *warps* of 32 threads. SM can support zero-overhead scheduling to switch between *warps* to hide long latency operations like off-chip memory access. The total number of concurrent *warps* reflects the *occupancy* of SM, which is determined by the physical resource limitations on chip. For more details of GPU programming, readers are referred to the NVIDIA GPU Programming Guide or online course materials [5].

Optimization on GPU. We summarize five guidelines to improve implementation efficiency on GPU, which will be used in Section IV.

- **G1** Reducing the arithmetic operation count. This is an algorithm level optimization. Reducing operation count may introduce side effects like breaking down regular data structure if not used properly.
- **G2** Reducing off-chip memory accesses. This can be achieved by improving data reuse in on-chip memory. A common strategy is "blocking": to organize the computation and data structure to better explore the data locality.
- **G3** Choosing appropriate memory types to optimally balance their pros and cons.
- **G4** Organizing global memory accesses in half *warps* in a coalesced manner when possible.
- **G5** Choosing optimal thread block size to balance impacts of *occupancy* and register utilization efficiency. Higher *occupancy* can better hide instruction latency, but it may reversely affect the overall performance if leading to worse utilization of register resources (e.g. causing a large number of register spills).

The first two guidelines **G1** and **G2** are related to first order analysis to identify whether the program is compute bound or memory bound. The theoretical upper bound for computation and memory access indicates

$$\frac{\#\text{of arithmetic ops}}{\text{processing time}} \leq 345.6\text{Gflop/s} \tag{1}$$

$$\frac{\#\text{of memory access (Bytes)}}{\text{processing time}} \leq 86.4\text{GB/s} \tag{2}$$

G3 addresses choosing the right type of memory for specific applications. The GTX 8800 offers various types of memory suited for different situations [5]. **G4** is important for improving memory bandwidth utilization efficiency. The highest bandwidth can be achieved when the global memory accesses are organized in a coalesced way, i.e., 16 threads in a half *warp* access 16 continuous data elements of 32-, 64- or 128-bit data types , and the starting address must be aligned. **G5** suggests tuning for the optimal thread block size to balance various factors for the best overall performance.

B. Related Work

Most existing stereo vision algorithms consist of four steps, as suggested by Scharstein and Szeliski [7]: (1) matching cost initialization; (2) cost aggregation ; (3) disparity optimization; and (4) disparity refinement. Stereo algorithms can be roughly classified into local and global approaches. Local algorithms use Winner-Take-All (WTA) strategy, simply taking the disparity level that minimizes the aggregation cost. Global algorithms apply energy minimization techniques to compute the optimal solution to a global energy function, which usually incorporates explicit smoothness assumptions. We categorize existing real-time or near real-time stereo systems into four major classes: local/global stereo on GPU/CPU.

Local stereo on GPU. Local approaches on a graphics processing unit (GPU) produce good quality disparity map at very fast speed. Gong et al. [4] discusses an interesting accuracy-speed trade-off of six cost aggregation approaches with WTA optimization under the real-time constraint on an ATI Radeon X800 graphics card. Their experiments show that a modified version of the adaptive weight window approach performs the best in terms of accuracy, running at about

TABLE I
SUMMARY OF REAL-TIME OR NEAR REAL-TIME STEREO SYSTEMS (QUANTITATIVE RESULTS ARE GIVEN LATER IN TABLE IV)

GPU	local ([4], [13], [14])	very fast, good accuracy
	global ([15], [16], [11], [3])	slower and better accuracy than local stereo on GPU
	proposed ESAW	faster than local stereo on GPU at comparable accuracy
	proposed ESMP	faster than global stereo on GPU at comparable accuracy
CPU	local ([8], [6], [10])	worse Pareto-efficiency in accuracy-speed trade-off space than local stereo on GPU
	global ([1], [2])	worse Pareto-efficiency in accuracy-speed trade-off space than global stereo on GPU

124 MDS on the ATI Radeon X800. The adaptive weight approach was originally proposed by Yoon and Kweon [17], which was very expensive in computation. Earlier real-time local stereo systems on the GPU include multi-resolution stereo by Yang, R., et al. [13], [14]. Other related work includes real time high quality stereo-based view synthesis systems on GPUs [20], [21].

Global stereo on GPU. Global stereo on the GPU is also extensively studied. Gong et al. [3] proposed a near real-time stereo based on ORDP (orthogonal reliability-based dynamic programming) on the ATI Radeon 9800 XT graphics card, running at about 20 MDS. Wang, L., et al. [11] proposed a real-time stereo algorithm on the ATI Radeon XL1800 graphics card. It integrated the adaptive weight aggregation along the vertical direction with dynamic programming optimization along horizontal scanlines. The disparity accuracy is slightly better than [4]. The system runs at about 52.8 MDS. Yang, Q., et al. [15] proposed a near real-time global stereo matching using hierarchical belief propagation on the NVIDIA Geforce 7900 GTX graphics card. It produces better accuracy than [11], but runs slower at about 17 MDS. Yang, Q., et al. [16] proposed a near real-time system on the NVIDIA Geforce 8800 GTX graphics card that incorporates color segmentation and plane fitting with belief propagation. The accuracy is further improved compared to [15]. The system runs at about 9.4 MDS. [24] propose an efficient implementation of dynamic programming approach using a recursive scheme, suitable for parallel stream computation model. [22] propose a near real time implementation of the semi-global matching algorithm in [23], running at about 9MDS for large image size and disparity range.

Local stereo on CPU. Several real-time local stereo systems on a general purpose CPU have been proposed. The Point Grey commercial stereo package can achieve 205 MDS on a 2.8GHz Intel PIV PC based on local window matching [6]. Veksler [10] proposed a fast stereo based on variable windows using integral images. Tombari et al. [8] presents a segmentation-based cost aggregation strategy that runs at 18.9 MDS on the Intel Core Duo 2.14 GHz CPU, achieving the best accuracy among existing near real-time local approaches on CPU. However, both accuracy and speed are worse compared to the real-time local stereo on GPU platforms [4], showing a certain gap between CPU/GPU processing power for stereo vision. In addition, Tombari et al. [9] classify the main cost aggregation approaches proposed in the literature based on both accuracy and processing speed on the Intel Core Duo 2.14 GHz CPU. Though implementation is not fully optimized, it gives an interesting overview picture of the trade-off between accuracy and computational complexity for cost aggregation methods.

Global stereo on CPU. Fast global stereo systems on a CPU are not common. Felzenszwalb and Huttenlocher [1] proposed a near real-time stereo system on the 2GHz Pentium IV based on loopy belief propagation, running at about 1.8 MDS. They propose several algorithm level optimization techniques. [15] also used those techniques in the GPU acceleration. Forstmann et al. [2] accelerated a dynamic programming based algorithm using MMX instructions, achieving about 100 MDS on an AthlonXP 2800+ 2.2G computer.

Summary. Table I gives an overview of the trade-off of various real-time stereo systems. Quantitative results are given later in Table IV. Generally speaking, stereo algorithms on CPU platforms can hardly match the Pareto-efficiency achieved on GPU platforms. Stereo systems based on global optimization methods like dynamic programming or belief propagation usually produce more accurate disparity map, at the cost of slower processing speed.

III. PROPOSED STEREO ALGORITHM

In this section, we propose two stereo algorithms: exponential step size adaptive weight (ESAW) and exponential step size message propagation (ESMP).

A. Exponential Step Size Adaptive Weight Algorithm

Exponential step size adaptive weight (ESAW) is an extension of the real-time adaptive weight approach in [4]. The main advantage of the proposed ESAW is to save arithmetic computation without degrading parallelism or accuracy. The algorithm in [4] is a simplification of the adaptive weight window cost aggregation originally proposed in [17]. We first briefly summarize the basic adaptive weight aggregation, and then explain the proposed ESAW.

Algorithm description. In cost aggregation, the matching cost of a pixel is the aggregated cost of all pixels in a surrounding support window of the center pixel. The basic idea of the adaptive weight approach is to adjust the per-pixel weight based on color dissimilarity and geometric relationship with the center pixel under consideration. Intuitively, a pixel is assigned a higher weight if it is closer in color and distance to the center pixel. Figure 1 second column illustrates examples of adaptive pixel weights with respect to the center pixel. The advantage of the adaptive weight approach is that it can preserve arbitrarily shaped depth discontinuities using a fixed window size. The disadvantage is that it performs worse than box filtering for heavy-textured areas, e.g., meadows.

Gong et al. [4] propose two simplifications over the basic adaptive weight to achieve real-time implementation on GPU.

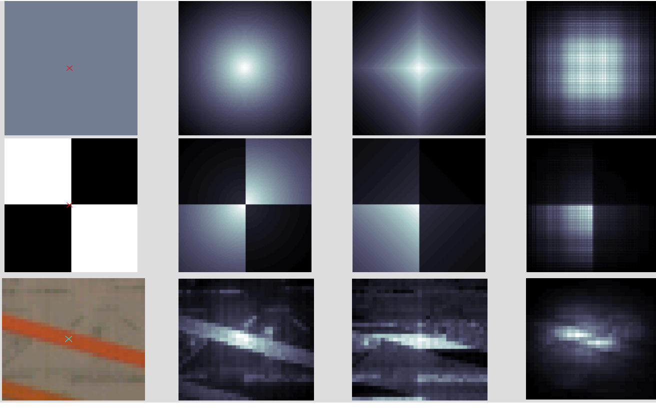


Fig. 1. From left to right: sample window, AW[17], RtAW [4] and ESAW.

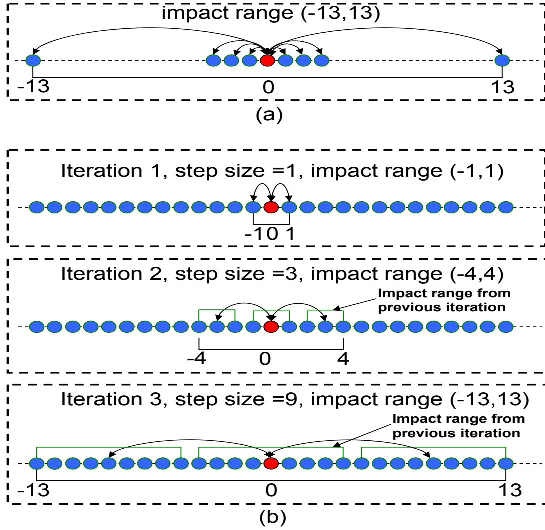


Fig. 2. A 1-D example of cost aggregation: (a) conventional aggregation used in [4], (b) exponential step size information propagation.

We name it RtAW in the following text. First, only weights in the reference view are used in cost aggregation. Second, instead of using a fixed window of size $N \times N$, a two pass approach is employed: the first pass aggregates cost along horizontal scanline, followed by a pass aggregating cost along vertical scanline. This reduces the arithmetic complexity from $O(N^2)$ to $O(N)$ per pixel.

We propose to use exponential step size in cost aggregation, which greatly saves the operation count without sacrificing data parallelism. We first explain the idea of exponential step size cost aggregation in the 1-D case. It takes $O(N)$ computations to aggregate the costs of all pixels within $r = \lfloor N/2 \rfloor$ offset to the center pixel along 1-D scanline using the direct aggregation method in [4]. Figure 2 (a) shows a simple example of aggregating pixels within range (-13, 13) needs computation on 27 pixels. Figure 2 (b) shows another way of aggregation with much less computation, achieved by 3 iterations. In each iteration, every pixel aggregates the costs of three pixels, itself and pixels at $-s$ and $+s$ offset. Offset s is set to 1, 3, 9 for three iterations. The "impact range" is defined as the largest pixel offset where the pixel matching cost is aggregated into the center pixel. After each iteration, the impact range grows, first from (-1, 1) to (-4, 4), then from (-4, 4) to (-13, 13). In this way, aggregating the matching costs of all pixels within range (-13, 13) just needs computation on $3 \times 3 = 9$ pixels.

Now we generalize the toy example. Assuming that the impact range after iteration $t-1$ is $-r(t-1)$ to $r(t-1)$, then the maximum step size $s(t)$ at iteration t is

$$s(t) = 2r(t-1) + 1$$

to avoid holes or gaps, which we don't expect to see because closer pixels are more correlated. With this step size, the impact range after iteration t becomes $-r(t)$ to $r(t)$, where

$$r(t) = 3r(t-1) + 1$$

With simple recursion, it can be derived that starting from $r(0) = 0$, the maximum step size and the impact range are

$$s(t) = 3^{t-1} \quad \text{and} \quad r(t) = (3^t - 1)/2$$

By using exponential step size, aggregating the costs of N pixels needs only $O(\log N)$ computations.

The idea can easily be extended to the 2-D case, by applying a vertical pass after a horizontal pass in each iteration. This is the proposed ESAW cost aggregation scheme. Using ESAW approach, aggregating the costs of $N \times N$ pixels is reduced to $O(\log N)$ computations per pixel.

The complete ESAW algorithm is summarized as following:

---

1. Initialize the matching costs at each pixel p at every disparity

$$C^0(p, d) = \lambda \min(\mathcal{I}_L(p_x, p_y) - \mathcal{I}_R(p_x - d, p_y), \tau) \tag{3}$$

2. Iterative cost aggregation

for $t = 1 : T$

(a) Compute offset:

$$s = \text{round}(b^{t-1}) \tag{4}$$

(b) **Aggregate the costs horizontally** of center pixel p at (x, y), p_l at $(x - s, y)$ and p_r at $(x + s, y)$:

$$\mathbf{C}^h(p) = \sum_{q \in \{p_l, p, p_r\}} \overline{w}(q, p) \mathbf{C}^{(t-1)}(q) \tag{5}$$

(c) **Aggregate the costs vertically** of center pixel p at (x, y), p_u at $(x, y - s)$ and p_d at $(x, y + s)$:

$$\mathbf{C}^t(p) = \sum_{q \in \{p_u, p, p_d\}} \overline{w}(q, p) \mathbf{C}^h(q) \tag{6}$$

end

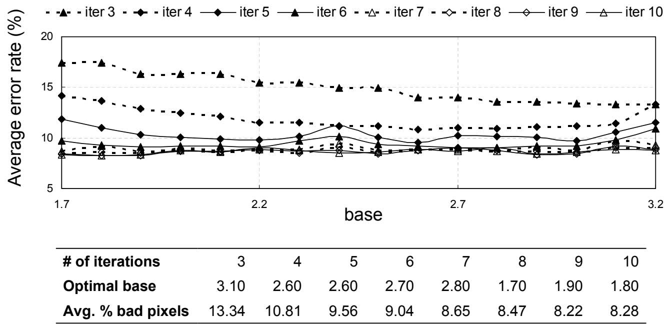
3. Choose the best disparity

$$d = \arg\min_d \mathbf{C}^t(p) \tag{7}$$

4. Post-processing disparity map using 3×3 median filter.

$\mathcal{I}(p_x, p_y)$ is the grayscale luminance of pixel p, by eliminating the hue and saturation. $\mathbf{C}^t(p)$ is a vector denoting all $C^t(p, d)$, which is the aggregated cost of pixel p after iteration t. $\mathbf{C}^h(p)$ is the intermediate horizontally aggregated cost. $\overline{w}$ is the normalized weight computed as in [17].

Algorithm analysis. Figure 3 shows the reconstruction accuracy of the ESAW algorithm, for varying number of iterations (3 to 10) and base b (1.5 to 3). The average error



# of iterations	3	4	5	6	7	8	9	10
Optimal base	3.10	2.60	2.60	2.70	2.80	1.70	1.90	1.80
Avg. % bad pixels	13.34	10.81	9.56	9.04	8.65	8.47	8.22	8.28

Fig. 3. Error rate of ESAW for varying b and iteration numbers.

rate is average percent of bad pixels (last column in Middlebury stereo evaluation online system) of all four benchmark datasets. The other parameters are empirically chosen

$$\gamma_c = 17,\ \gamma_p = 36,\ \tau = 12.$$

γ_c, γ_p are parameters used for computing the adaptive weights as in [17]. Since the computing time grows linearly with the number of iterations, we choose an optimal base b giving the best accuracy for each iteration number, as summarized in Figure 3. Sensitivity of the average error rate with respect to each parameter is shown in Figure 4.

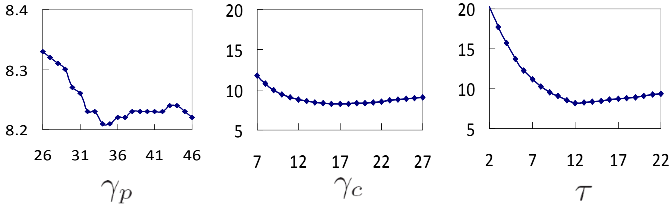


Fig. 4. Sensitivity analysis of ESAW parameters.

It is observed that the accuracy improves with the number of iterations, though improvement gets very marginal after iteration 7.

Our aggregation scheme does not produce exactly the same result as [4] or [17]. But all three approaches have one thing in common: in general, they tend to assign higher weights to closer pixels. Figure 1 shows the weights for three cases: 1) window has constant color; 2) window has sharp color change; 3) a real window from Tsukuba image. In all three approaches, pixels assigned high weights are close to the center in terms of both color and geometric distance. That's why they generate disparity maps of similar quality.

B. *Exponential Step Size Message Propagation Algorithm*

Motivation. One basic assumption in the local cost aggregation is that pixels within the window have the same disparity, i.e. the local window is frontal plane. The optimal solution is

$$d = \arg\min_{d} \sum_{q} \overline{w}(q,p) C^0(q,d) \tag{8}$$

If we relax the constraint of frontal plane to allow small disparity variation within the window, the optimization problem can be formulated as

$$d_p = \arg\min_{d_p} \sum_{q} \overline{w}(q,p)(C^0(q,d_q) + V(d_p - d_q)) \tag{9}$$

$$V(x) = \min(c|x|, \eta) \tag{10}$$

$d_q (q \neq p)$ can take arbitrary disparity values. $V(x)$ is exactly the smoothness term widely used in belief propagation based stereo [25], [1], penalizing disparity changes from p to q. The solution to equation (9) can be computed as:

$$M(q,d) = \min_{d_q}(C^0(q,d_q) + V(d_q - d)) \tag{11}$$

$$d_p = \arg\min_{d_p} \overline{w}(q,p) \sum_{q} M(q,d_p) \tag{12}$$

If we compare equation (8) and (12), the difference is that (12) aggregates "message" $M(q,d)$ and (8) aggregates "cost" $C^0(q,d)$. Equation (11) is about how to map $C^0(q,d)$ to $M(q,d)$, which is exactly the min-sum message computation in [1]. This mapping has a "smoothing" effect on $C^0(q,d)$. $M(q,d)$ is the lower envelop of cones rooted at each discrete disparity level and the constant truncation value. A simple example is illustrated in Figure 5. For more details, readers are referred to [1].

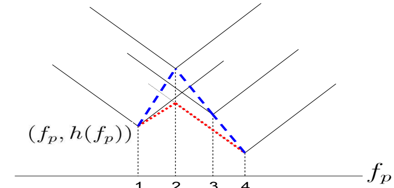


Fig. 5. An illustration of min-sum computation (without truncation). Output is the lower envelop of four Cones rooted at $(f_p, h(f_p))$. Dashed blue curve shows $h(f_p)$, dotted red curve shows the lower envelop.

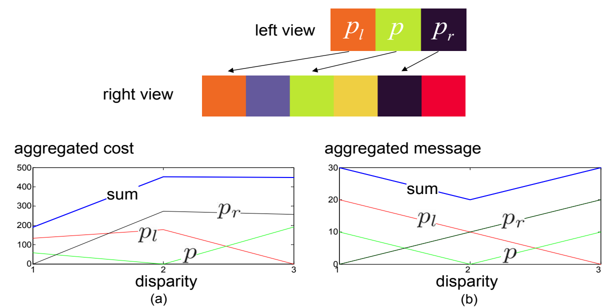


Fig. 6. A synthesized example showing difference of aggregating matching costs and aggregating messages. Ground truth disparity of pixel p is 2. (a) blue curve shows the result of aggregating matching costs for p, where minimum is reached at disparity 1. (b) blue curve shows the result of aggregating messages for p, where minimum is reached at true disparity 2. Messages are computed from matching costs using equation (11). In this example, $V(x) = 10|x|$.

In Figure 6, we use a toy example to show the difference between aggregating matching costs and aggregating messages. Assuming true disparities of pixel p_r, p, p_l are 1, 2, 3 respectively, and totally there are only three disparity levels. In Figure 6 (a), we show the matching cost of each pixel at all 3 disparity levels. The blue line is the summation of costs

of all pixels. In Figure 6 (b), we show the message of each pixel at all 3 disparity levels. Messages are computed from matching costs using equation (11). Blue curves in Figure 6 (a) and (b) show the aggregation results. By aggregating messages instead of matching costs, the blue curve in Figure 6 (b) can reach minimum at the true disparity. This is because in highly textured areas, pixels may have large matching errors at disparity levels deviating a little from the true disparity. In this example, p_l, p_r have large matching costs at disparity 2, resulting in a large aggregation cost of p at disparity 2. Aggregating messages tend to have a "smoothing" effect on the matching costs as illustrated in Figure 5, therefore it is more robust to small variation in disparity.

Figure 7 shows a more complicated synthesized example. Assuming that disparity increases from left to right linearly from 0 to 64 pixels, the right view is synthesized from the left view and the given disparity map. The second row shows the disparity results of ESAW and ESMP (parameters are the same as those in Section V). The disparity of ESMP is much smoother than the disparity of ESAW. The third row shows error pixels (whose estimation error > 1 pixel) of two algorithms. In this case, the error rate for ESAW is $2\times$ of the error rate for ESMP. ESMP demonstrates to be much more robust than ESAW when true disparity deviates from frontal plane assumption.

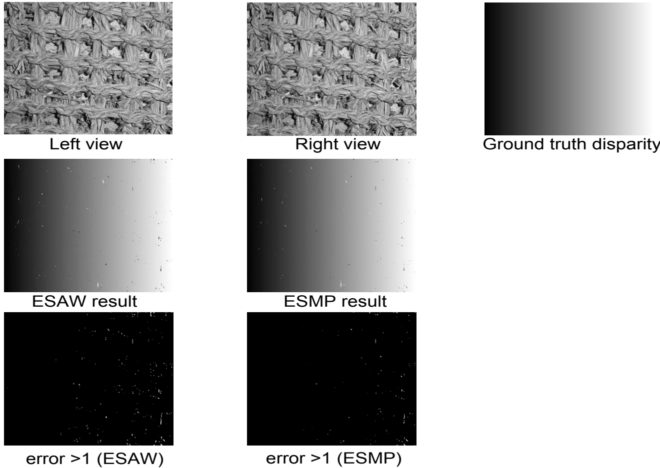


Fig. 7. A synthesized example to demonstrate that ESMP produces a more accurate disparity map than ESAW when true disparity is not a frontal plane.

Algorithm description. As an extension of the ESAW algorithm, ESMP shares the same steps 1, 3 and 4 of ESAW. The difference lies in step 2, as shown in the following:

2. Iterative cost aggregation

for $t = 1 : T$

(a) Compute offset as in equation (4).

(b) **Map costs to messages** (`cost2msg`):

$$M(p,d) = \min_{d'} (V(d-d') + C^{(t-1)}(p,d')) \quad (13)$$

(c) **Aggregate the messages horizontally**:

$$\mathbf{C}^h(p) = \sum_{q \in \{p_l, p, p_r\}} \overline{w}(q,p)\mathbf{M}(q) \quad (14)$$

(d) **Map costs to messages** (`cost2msg`):

$$M(p,d) = \min_{d'} (V(d-d') + C^{(h)}(p,d')) \quad (15)$$

(e) **Aggregate the messages vertically**:

$$\mathbf{C}^t(p) = \sum_{q \in \{p_u, p, p_d\}} \overline{w}(q,p)\mathbf{M}(q) \quad (16)$$

end

We use the fast min-sum algorithm as in [1] to reduce the complexity from $O(\ell^2)$ to $O(\ell)$ (ℓ is the total number of disparity levels). The fast algorithm to map $C(p,d)$ to $M(p,d)$ is:

$$\mathbf{M}(p) = \mathbf{C}(p) \quad (17)$$

$$h = \min_d M(p,d) + \eta \quad (18)$$

for $d = 1 : 1 : \ell - 1$

$$M(p,d) = \min(M(p,d-1) + c, M(p,d)) \quad (19)$$

end

$$M(p,\ell-1) = \min(M(p,\ell-1), h) \quad (20)$$

for $d = \ell - 2 : -1 : 0$

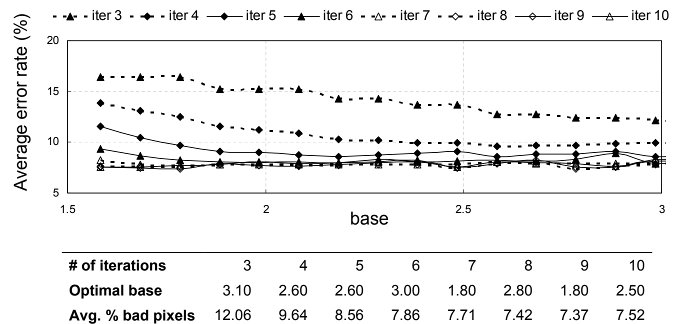
$$M(p,d) = \min(M(p,d+1) + c, M(p,d), h) \quad (21)$$

end

Algorithm analysis. Figure 8 shows the reconstruction accuracy of the ESMP algorithm, for varying number of iterations (3 to 10) and base b (1.6 to 3.1). The average error rate is average percent of bad pixels (last column in Middlebury stereo evaluation online system) of all four benchmark datasets. The other parameters are empirically chosen

$$\gamma_c = 18,\ \gamma_p = 29,\ \tau = 17,\ \lambda = 0.15,\ c = 1,\ \eta = 0.0375 d_m,$$

where d_m is the maximum disparity value. Sensitivity of the average error rate with respect to each parameter is shown in Figure 9.



# of iterations	3	4	5	6	7	8	9	10
Optimal base	3.10	2.60	2.60	3.00	1.80	2.80	1.80	2.50
Avg. % bad pixels	12.06	9.64	8.56	7.86	7.71	7.42	7.37	7.52

Fig. 8. Error rate of ESMP for varying b and iteration numbers.

Compared to standard belief propagation (BP) based global optimization, ESMP has three differences:

- Global optimization optimizes a global energy function. ESMP instead aggregates messages coming from a predefined bounded support like other local aggregation algorithms.

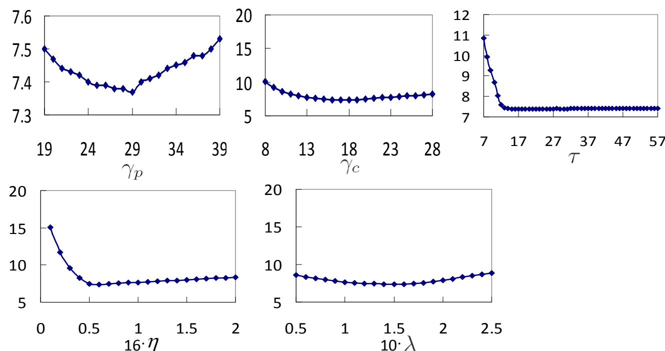


Fig. 9. Sensitivity analysis of ESMP parameters.

- ESMP aggregates information using an exponential step size, while the standard BP always uses step size 1.
- Messages in ESMP are isotropic, independent of the direction where the message is sent to, thus reducing the memory to $1/4$ of what's needed in the standard BP. The huge message storage requirement (about 1GB for 640×480 with 200 disparities in the standard BP) presents great challenge for embedded systems and processors exhibiting memory bandwidth limitations. Other solutions include compression techniques [19] and search space reduction [12]. However, those solutions either require an additional coding/decoding process, or making data structure less regular and parallelization on multi-core platforms more difficult. ESMP, on the other hand, keeps the highly parallel data structure.

IV. Mapping Algorithms to the GPU

On a high level, ESAW and ESMP are highly data parallel algorithms suitable to be implemented on a GPU architecture. However, there are still various choices to be made in code optimization to achieve the best performance. Following questions are what we found most related to the performance. First, which kind of off-chip memory should be used for storing the costs after each aggregation pass? Second, how to organize off-chip memory accesses to improve the bandwidth utilization efficiency? Third, whats the optimal thread block size? Fourth, where the intermediate results should be stored and when to transfer data to off-chip memory? We start from a straight forward implementation of ESAW, then gradually optimize the code by answering the above questions.

A. Baseline implementation

In ESAW, computation *kernels* include:

- `rgb2grey` (compute luminance from a color image);
- `init_cost` (initialize matching cost in equation (3));
- `aggr_H` (horizontal aggregation in equation (5));
- `aggr_V` (vertical aggregation in equation (6));
- `select_disparity` (choose the best disparity in equation (7));
- `median_filter` (post-processing).

In each *kernel*, the same computation is done on every pixel. Naturally, the image is segmented into a *grid* of blocks, each corresponding to one thread block. The block size $b_h \times b_w$ is adjustable, but it must be a multiple of 32 to be divided into multiple *warps*. Block width b_w should be a multiple of 16 to allow coalesced memory access pattern. In our baseline, block size is fixed to be 4×32. After each aggregation, the aggregation cost is copied into texture memory to save management of out-of-boundary addressing.

For ESAW at iteration 9 on Teddy dataset, the baseline implementation takes about 99.9ms. The time spent on each *kernel* function, memory copy to or from the host CPU, and memory copy time on the GPU device are illustrated in Figure 12 “ESAW baseline”. About 35% of the time is spent on memory copy on the GPU device, and 61% of the time is spent on horizontal/vertical iterative aggregation.

B. Optimization techniques

Texture vs. global memory. Both texture memory and global memory can be used for storing the costs. We simply substitute texture memory with global memory without any optimization on memory access pattern. The overall processing time increases to 113.9ms (Figure 12 “ESAW global memory”). The memory copy time on the GPU is much reduced, but `aggr_H` *kernel* is slowed down by about 4 times. The reason is that 75% of load instructions are un-coalesced. It is worth noticing that *kernel* `aggr_V` is sped up by $1.59\times$. This is because memory access pattern for `aggr_V` is naturally coalesced. For the same coalesced memory transactions, global memory is usually faster than texture memory.

Coalescing memory accesses. To further improve the performance, we organize global memory accesses into coalesced transactions. The reason for the un-coalesced loads in `aggr_H` is that the offset value s in equation (4) may not be a multiple of 16. This will violate the starting address alignment requirement in the coalesced access pattern. Figure 10 illustrates the case when $s = 3$. Each thread needs to read 3 pixels at $(x-3, y)$, (x, y), $(x+3, y)$. Though 16 pixels at (x, y) can be accessed in the coalesced pattern, the left and right offset pixels cannot be loaded in a coalesced way.

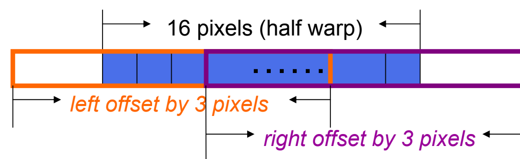


Fig. 10. An example of un-aligned memory access pattern in horizontal aggregation, when $s = 3$.

The solution for coalescing memory accesses is to use the on-chip shared memory, as illustrated in Figure 11. Chunks of pixels are first read from global memory into the on-chip shared memory in a coalesced way, and then threads load data from the shared memory for computation. Accessing data in the shared memory does not require address alignment and is as fast as registers. Coalescing memory accesses reduces the processing time to 53.5ms (Figure 12 “ESAW coalesced”).

Tuning thread block size. Tuning block size involves intricate trade-off between per block efficiency and occupancy.

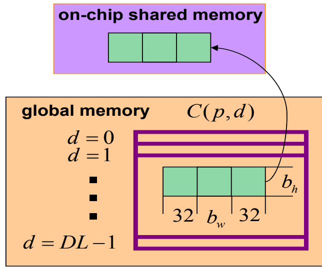


Fig. 11. Illustration of bringing data from the off-chip memory to the on-chip shared memory. Every block of size $b_h \times b_w$ brings pixels inside this block, and extends 32 pixels to the left and right.

Total number of bytes loaded from the global memory in one horizontal pass is

$$(9 + (1 + 64/b_w)4DL)hw$$

where h, w are image height and width, ℓ is the number of disparity levels. So increasing block width b_w helps reduce off-chip memory accesses. But increasing b_w may decrease the *occupancy* because each *thread block* needs more resources. We search for the best block size configuration that gives the best performance. With tuning, computation time reduces to 43.4ms (Figure 12 "ESAW tune block size"). `aggr_H` and `aggr_V` count for about 92% of the total processing time.

So far, we have discussed about optimization for ESAW. Next we will discuss implementation of ESMP.

Reorganizing data accesses. The most natural way of extending ESAW to ESMP is to add one more *kernel* function `cost2msg` that maps the costs to messages using equations (18)–(21). A straight-forward implementation for `cost2msg` *kernel* takes 90.5ms. `cost2msg` contains a forward pass (equation (18)–(19)) and a backward pass (equation (20)–(21)). We found that performance for `cost2msg` *kernel* is 22.2 Gflop/s, and bandwidth utilized is 59.3GB/s. The `cost2msg` *kernel* is memory bound.

It is not practical to use the on-chip shared memory to alleviate off-chip memory accesses, because each thread needs 4ℓ bytes space to store cost values. This means each *warp* needs 8k bytes on-chip memory when $\ell = 64$, so at most 2 *warps* can run concurrently on one SM, leading to an extremely low *occupancy* of 0.08. Also for stereo requiring large number of disparity levels, this technique is not scalable.

We propose a more practical solution. At the end of each aggregation, costs are written back to the global memory, and then they are read from the global memory at the beginning of `cost2msg` *kernel*. This motivates us to integrate the forward pass of `cost2msg` with the aggregation *kernel*, which slightly increases the complexity of the aggregation *kernel*, but saves half of the global memory accesses in `cost2msg`. We rename the reorganized *kernels* as `aggr_H+`, `aggr_V+` and `cost2msg-`. With this technique, *kernel* execution time for `cost2msg` is sped up by about 2×. The total processing time reduces to 65.2 ms for ESMP (Figure 12 "ESMP reorganized").

In Table II, we summarize the arithmetic operation count and bandwidth used for the most time-consuming *kernels*: `aggr_H`, `aggr_V` in "ESAW tune block size", and `aggr_H+`, `aggr_V+`, `cost2msg-` in "ESMP reorganized". Clearly the performance is memory bound. Figure 12 shows the time spent on each *kernel* in the above implementations on Teddy image dataset at iteration 9.

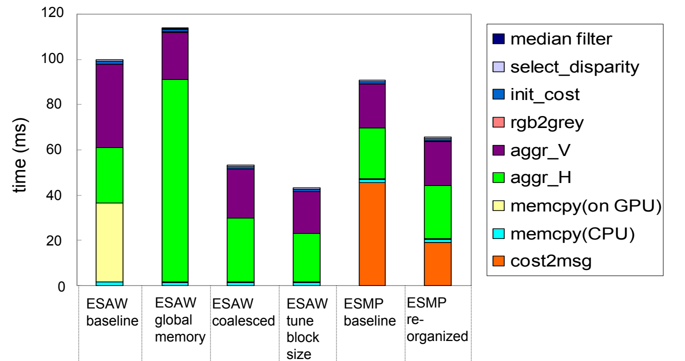


Fig. 12. Profile of execution time of different implementations on Teddy dataset, at iteration 9. (Best view in color)

TABLE II
COMPUTATIONAL AND COMMUNICATIONAL COST FOR THE MOST TIME-CONSUMING KERNELS.

	arithmetic operations	off-chip read(B)	off-chip write(B)	op/s (Gflop/s)	BW (GB/s)
`aggr_H`	$(33 + 5\ell)hw$	$(9 + 6\ell)hw$	$4\ell hw$	19.4	34.5
`aggr_V`	$(33 + 5\ell)hw$	$(9 + 12\ell)hw$	$4\ell hw$	22.7	63.9
`aggr_H+`	$(33 + 8\ell)hw$	$(9 + 6\ell)hw$	$4\ell hw$	28.7	33.4
`aggr_V+`	$(33 + 8\ell)hw$	$(9 + 12\ell)hw$	$4\ell hw$	33.9	62.8
`cost2msg-`	$3\ell hw$	$4\ell hw$	$4\ell hw$	21.6	57.7

h, w are image height and width. ℓ is the total number of disparity levels.

Figure 13 shows processing time of our final version "ESAW tune block size" and "ESMP reorganized" for the four benchmark datasets, for iteration number 3 to 10.

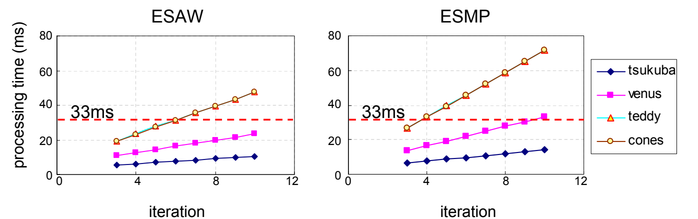


Fig. 13. Execution time of "ESAW tune block size" and "ESMP reorganized" on four datasets, for 3 to 10 iterations. Red dashed line (33ms) shows where real-time performance can be achieved.

V. EXPERIMENTAL RESULTS

In this section, we compare both accuracy and system throughput with existing stereo systems. Our implementation uses all optimization techniques discussed in Section IV. The accuracy is measured by the average percent of bad pixels for non-occluded, all, and discontinuous areas on all four benchmark datasets. The throughput is reported as average MDS on Teddy and Cones dataset (MDS for Tsukuba and

Venus is bit lower due to the higher overhead for small image size).

Comparison to RtAW[4] and AW[17]. Figure 14 shows trade-off between the average error rate and algorithm complexity in ESAW, RtAW[4] and original window based AW[17]. It is worth noticing that in RtAW and AW after fixing γ_c, γ_p and τ, the window size decides both algorithm complexity and accuracy. However in ESAW after fixing γ_c, γ_p and τ, the accuracy depends on iteration number T and base b. The algorithm complexity depends on T. It can be seen in Figure 3 that the optimal base is not the one having the largest impact range or the effective window size, meaning in ESAW the effective window size itself cannot decide the accuracy. Therefore, a fair comparison is to compare the accuracy complexity trade-off.

In Figure 14, parameters for ESAW are the same as in Figure 3. γ_c, γ_p and τ are the same for ESAW and RtAW. For RtAW, we show results of varying window size in {17, 33, 65, 97}. For AW, we use the code provided online[18], and show results of varying window size in {17, 33}. Program is aborted due to lack of memory on a 4G RAM machine for window size 65 and 97. Algorithm complexity is estimated by arithmetic operations(ops) per pixel per disparity level. For ESAW it is about $2T \cdot 5$, because 5 ops are needed to compute Eq. (5) or (6). For RtAW it is about $2 \cdot (2W_s - 1)$ (W_s is the window size), because $2W_s - 1$ ops are needed to compute horizontal/vertical aggregation. For AW it is about $5W_s^2 - 1$ to compute window-based aggregated error, according to Eq. (7) in [17]. Clearly complexity of ESAW is the lowest among three algorithms at comparable accuracy. We also implement message aggregation for RtAW. Results of ESMP and RtAW+MP are also shown. Parameter setting for ESMP is the same as in Figure 8. γ_c, γ_p, τ, λ and η are the same for ESMP and RtAW+MP. For both ESAW and RtAW, we see message aggregation improves accuracy at the cost of higher complexity compared to direct cost aggregation.

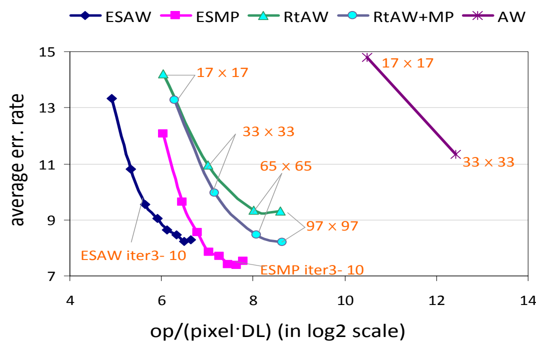


Fig. 14. Accuracy-complexity trade-off of ESAW, ESMP, RtAW, RtAW+MP, and AW. x-axis shows arithmetic operation count per pixel per disparity level. y-axis shows the average error rate. Parameter settings are stated in text.

Comparison to other real-time stereo systems. Figure 15 plots error rates for non-occluded and discontinuous areas versus the normalized processing time in $\log_2$ scale (ns per disparity evaluation, which is the reciprocal of MDS), for real-time stereo systems on the GPU. The proposed ESAW and ESMP suggest a number of Pareto-optimal configurations in the accuracy-speed trade-off space.

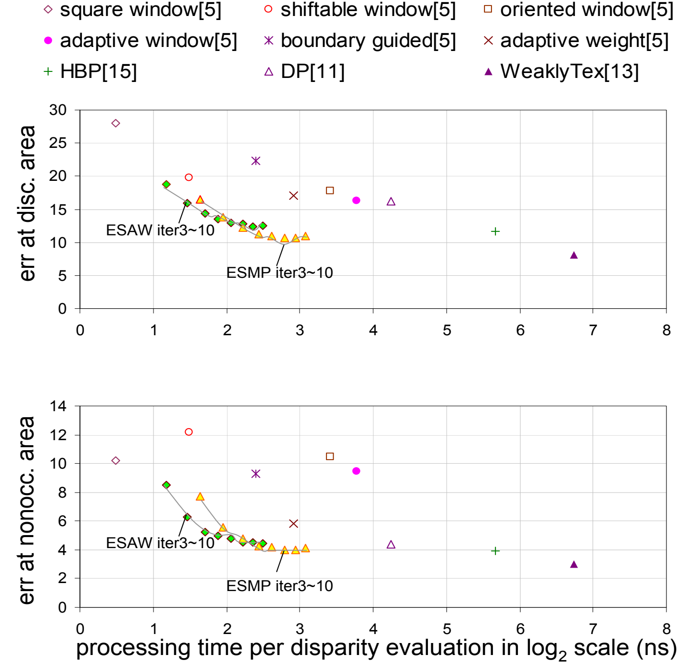


Fig. 15. Error rate versus processing time for real-time stereo on the GPU.

TABLE III
PARAMETER SETTINGS IN ESAW AND ESMP.

ESAW	iter 5	$\gamma_c = 17,\ \gamma_p = 36,\ \tau = 12,\ \ b = 2.60$
	iter 9	$\gamma_c = 17,\ \gamma_p = 36,\ \tau = 12,\ \ b = 1.90$
ESMP	iter 8	$\gamma_c = 18,\ \gamma_p = 29,\ \tau = 17,\ \lambda = 0.15,\ c = 1,$ $\eta = 0.0375d_m,\ b = 2.80$

Table IV shows quantitative comparison results using four benchmark datasets. Parameter settings are listed in Table III. Disparity maps are shown in Figure 16. In terms of accuracy, ESMP at iteration 9 outperforms all other real-time or near real-time stereo systems on GPU except for WeaklyTex in [16]. WeaklyTex incorporates color segmentation and plane fitting, and it is much slower than ESMP. At comparable accuracy, the proposed systems can achieve much higher throughput compared to existing stereo systems. We also show C code implementation result of ESAW for reference on an off-the-shelf Core 2 Duo desktop CPU (Intel E6750 2.66GHz) with 2GB memory. Please note this result is based on single-threaded scalar code without full optimizations, which probably is much slower than the best code (fully optimized, using multiple threads and SSE vector instructions) possible.

Comparison of implementation efficiency. It is important to note that some previous works are not implemented on exactly the same platform. For a fair comparison across different platforms, we made our best effort to compare implementation efficiency with RtAW (adaptive weight 33 × 33 in [4]) and HBP ([15]), as shown in Table V. The estimation of memory access is a lower bound, based on the assumption that in each pass cost values are at least read and write once. Compared to ESAW, RtAW achieves comparable computational efficiency

TABLE IV
ACCURACY-SPEED COMPARISON OF REAL-TIME OR NEAR REAL-TIME STEREO SYSTEMS. THE DATASET AND ERROR RATE ARE THE SAME AS MIDDLEBURY ONLINE EVALUATION STEREO SYSTEM[26].

			Tsukuba			Venus			Teddy			Cones			Err	MDS
			vis	all	disc	vis	all	disc	vis	all	disc	vis	all	disc		
GPU	local	ESMP(iter 8)	1.4	1.9	7.1	0.4	1.0	2.7	8.6	15.2	19.5	5.6	12.5	13.2	7.42	144.4
		ESAW(iter 9)	1.9	2.5	9.7	1.0	1.7	6.9	8.5	14.2	18.7	6.6	12.7	14.4	8.2	194.8
		ESAW(iter 5)	1.4	2.4	7.1	1.6	2.6	12.9	9.4	16.0	19.4	8.6	15.6	17.9	9.6	307.2
		RtAW([4])	2.3	3.6	11.2	3.6	4.6	19.8	10.9	18.8	23.2	5.9	14.3	13.8	11.0	124.1
		RealTime([14])	9.7			15.7										117.0
	global	WeaklyTex([16])	1.0	1.8	5.3	0.2	0.5	1.7	6.7	12.1	14.7	4.2	10.7	10.6	5.8	9.4
		HBP([15])	1.5	3.4	7.9	0.8	1.9	9.0	8.7	13.2	17.2	4.6	11.6	12.4	7.7	17.0
		DP([11])	2.1	4.2	10.6	1.9	3.0	20.3	7.2	14.4	17.6	6.4	13.7	16.5	9.8	52.8
		ORDP([3])	1.4		7.4	2.4		13.5								20.0
CPU	local	ESAW(iter 9)	1.9	2.5	9.7	1.0	1.7	6.9	8.5	14.2	18.7	6.6	12.7	14.4	8.2	2.3
		ESAW(iter 5)	1.4	2.4	7.1	1.6	2.6	12.9	9.4	16.0	19.4	8.6	15.6	17.9	9.6	4.2
		aggregate([8])	3.0	4.4	13.2	3.5	4.6	25.5	10.7	17.5	23.4	4.9	12.7	11.3	11.2	18.9
		Integral([10])	2.4		12.2	1.2		13.4								< 1
	global	BP([1])	1.9	3.8	10.1	1.2	2.2	15.6	23.1	30.9	33.8	20.6	27.6	29.0	16.6	1.8
		RealTimeDP([2])	2.9		15.6	6.4		25.3								100.0
		AW([17])	1.4	1.9	6.9	0.7	1.2	6.1	7.9	13.3	18.6	4.0	9.8	8.3	6.7	< 0.1

(op/s). We believe this is due to the strong data locality in the RtAW algorithm. Another point worth noticing is that the operation count of ESAW at iteration 4 is only about 30% of RtAW, but they generate disparity maps of the same quality.

TABLE V
COMPUTATIONAL AND COMMUNICATIONAL EFFICIENCY COMPARISON.

	op/s (Gflop/s)	BW used (GB/s)	# of ops (G)	memory accesses(B)
ESMP iter5	28.0	50.5	1.00	1.80
ESAW iter4	20.9	48.1	0.39	0.90
RtAW	20.3	2.1†	1.30	0.14†
HBP	4.3	0.5†	2.55	0.64†

Note: ESMP and ESAW are run on GTX 8800, with peak performance of 350 Gflop/s and BW 86.4 GB/s; RtAW[4] is run on ATI Radeon X800, with peak performance of 200 Gflop/s and BW 35.8 GB/s; HBP [15] is run on Geforce 7900 GTX, with peak performance of 255 Gflop/s and BW 51.2 GB/s. †estimation is a lower-bound.

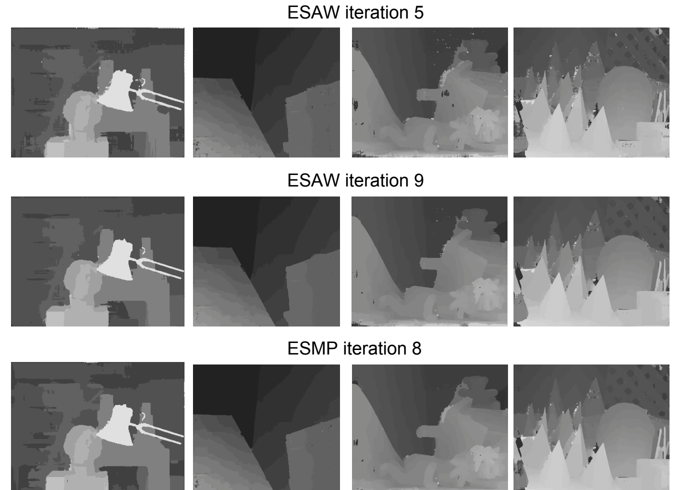


Fig. 16. Disparity map results.

VI. CONCLUSIONS

In this paper, we propose a high performance stereo system based on hardware-aware software design concept. Our system consists of two new algorithms: exponential step size adaptive weight (ESAW) and exponential step size message propagation (ESMP). ESAW can effectively reduce the operation count from $O(N)$ to $O(\log N)$ per pixel, where N is the aggregation window size. ESMP extends ESAW to incorporate the smoothness term, thus can better model non-frontal planes.

We also discuss various choices in code optimization. Instead of doing optimization in an 'ad hoc' manner, we analyze the trade-offs and bottleneck in the implementation to fully understand the efficiency of our code.

With the fast evolvement of computer architecture, an interesting future research direction may be to investigate hardware-software co-design for real time stereo.

ACKNOWLEDGMENT

We would like to thank Prof. Minglun Gong and Peter Milder for their helpful suggestions and discussions.

REFERENCES

[1] P. F. Felzenszwalb, and D. P. Huttenlocher, Efficient Belief Propagation for Early Vision, *International Journal of Computer Vision*, 2006
[2] S. Forstmann, J. Ohya, Y. Kanou, A. Schmitt, and S. Thuering., Real-time stereo by using dynamic programming, *Proc. of CVPR Workshop on Real-time 3D Sensors and Their Use*, 2004
[3] M. L. Gong and Y.-H. Yang , Near real-time reliable stereo matching using programmable graphics hardware, *Proc. of CVPR*, 2005
[4] M. L. Gong, R. Yang, L. Wang, and M. W. Gong, A Performance Study on Different Cost Aggregation Approaches Used in Real-Time Stereo Matching, *International Journal of Computer Vision*, 2007
[5] W. M. Hwu and D. Kirk, UIUC ECE 498: Programming Massively Parallel Processors
[6] Point Grey Reseach Inc., *Triclops Library Performance on Stereo Vision*, `http://www.ptgrey.com/support/kb/data`
[7] D. Scharstein and R. Szeliski, A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, *International Journal of Computer Vision*, 2002
[8] F. Tombari, S. Mattoccia, L. D. Stefano, and E. Addimanda, Classification and performance evaluation of different aggregation costs for stereo matching, *Proc. of CVPR*, 2008
[9] F. Tombari, S. Mattoccia, L. D. Stefano, and E. Addimanda, Near real-time stereo based on effective cost aggregation, *ICPR*, 2008

[10] O. Veksler, Fast variable window for stereo correspondence using integral images, *Proc. of CVPR*, 2003
[11] L. Wang, M. Liao, M. L. Gong, R. Yang and D. Nister, High-Quality Real-Time Stereo Using Adaptive Cost Aggregation and Dynamic Programming, *3DPTV*, 2006
[12] L. Wang, H. Jin, and R. Yang, Search Space Reduction for MRF Stereo, *Proc. of ECCV*, 2008
[13] R. Yang and M. Pollefeys, Multi-resolution real-time stereo on commodity graphics hardware, *Proc. of CVPR*, 2003
[14] R. Yang, M. Pollefeys, and S. Li, Improved real-time stereo on commodity graphics hardware, *Proc. of CVPR Workshop on Real-time 3D Sensors and Their Use*, 2004
[15] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao and D. Nister, Real-time Global Stereo Matching Using Hierarchical Belief Propagation, *BMVC*, 2006
[16] Q. Yang, C. Engels and A. Akbarzadeh, Near real-time stereo for weakly-textured scenes, *BMVC*, 2008
[17] K. J. Yoon and I. S. Kweon, Adaptive Support-Weight Approach for Correspondence Search, *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2006
[18] `http://vision.middlebury.edu/stereo/code/`
[19] T. Yu, R. S. Lin, B. Super, and B. Tang, Efficient message representations for belief propagation, *Proc. of ICCV*, 2007
[20] S. Rogmans, J. Lu, P. Bekaert, and G. Lafruit, Real-Time Stereo-Based View Synthesis Algorithms: A Unified Framework and Evaluation on Commodity GPUs, *Signal Processing: Image Communication, Special Issue on Advances in Three-Dimensional Television and Video*, 2009
[21] J. Lu, S. Rogmans, G. Lafruit, and F. Catthoor, Stream-Centric Stereo Matching and View Synthesis: A High-Speed Image-Based Rendering Paradigm on GPUs, *Transactions on Circuit Systems and Video Technology*, 2009
[22] I. Ernst and H. Hirschmuller, Mutual Information Based Semi-Global Stereo Matching on the GPU, *ISVC, Part I*, LNCS 5358, 2008
[23] H. Hirschmuller, Stereo processing by semi-global matching and mutual information, *LIEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008
[24] C. Zach, M. Sormann, K. Karner, Scanline Optimization for Stereo On Graphics Hardware, *3DPTV*, 2006
[25] J. Sun, N.-N. Zheng, and H.-Y. Shum, Stereo Matching Using Belief Propagation, *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2003
[26] D. Scharstein and R. Szeliski, *Middlebury Stereo Evaluation - Version 2*, `http://vision.middlebury.edu/stereo/eval/`