Real-Time Accurate Stereo with Bitwise Fast Voting on CUDA

Ke Zhang\textsuperscript{1,2,*}, Jiangbo Lu\textsuperscript{1,2}, Gauthier Lafruit\textsuperscript{1}, Rudy Lauwereins\textsuperscript{1,2}, Luc Van Gool\textsuperscript{2}
Interuniversity Micro-Electronics Center (IMEC), Kapeldreef 75, B-3001 Leuven, Belgium\textsuperscript{1}
Katholieke Universiteit Leuven, Leuven, Belgium\textsuperscript{2}
*\texttt{zhangke@imec.be}

Abstract

This paper proposes a real-time design for accurate stereo matching on Compute Unified Device Architecture (CUDA). We adopt a leading local algorithm for its high data parallelism. A GPU-oriented bitwise fast voting method is proposed to effectively improve the matching accuracy, which is enormously faster than the histogram-based approach. The whole algorithm is parallelized on CUDA at a fine granularity, efficiently exploiting the computing resources of GPUs. On-chip shared memory is utilized to alleviate the latency of memory accesses. Compared to the CPU counterpart, our design attains a speedup factor of 52. With high matching accuracy, the proposed design is still among the fastest stereo methods on GPUs. The advantages of speed and accuracy advocate our design for practical applications such as robotics systems and multiview teleconferencing.

1. Introduction

Stereo matching has been intensively studied as an important computer vision topic, with lots of work surveyed in \cite{32}. Evaluated in the stereo benchmark \cite{11}, state-of-the-art algorithms have dramatically improved the matching accuracy. However, these algorithms are usually computationally complicated and fast stereo matching with high accuracy is still challenging. Since real-time performance is crucial to applications such as robot navigation and multiview teleconferencing, many research efforts are attracted to tackle the challenge.

To attain fast speed, stereo on graphics processing units (GPUs) is an attractive trend, as a successful exemplar of computer vision on GPUs \cite{3}. Utilizing the horsepower of massive parallel processors, GPUs are effective to accelerate stereo algorithms by exploiting their potential parallelism. Several recent methods have reached fast speed on GPUs while maintaining matching quality \cite{5}, \cite{3}, \cite{6}. Yang et al. \cite{5} implemented a belief propagation (BP) approach. Though it achieved high matching accuracy, large memory consumption could impede its prevalence in matching high-resolution images with a large disparity range. Wang et al. \cite{3} presented an approach based on dynamic programming (DP). Though the streaking artifacts associated to the traditional DP are reduced in \cite{3}, the quality at depth-discontinuity regions is still not satisfying. Lu et al. \cite{6} obtained fast speed on a GPU using a cross-based local approach \cite{15}. Excluding the refinement stage of \cite{15}, Lu et al.’s method \cite{6} cannot handle occluded regions and large homogeneous regions accurately.

In terms of GPU architectures, we have witnessed a rapid evolution, migrating GPUs from a strict pipelined task-parallel architecture to a unified data-parallel programmable unit \cite{4}. As a modern GPU architecture, Compute Unified Device Architecture (CUDA) \cite{2} is supported by Nvidia GPUs for general-purpose parallel computing. Compared to traditional GPUs, CUDA provides more flexibility in managing and utilizing GPU resources while maintaining huge computational power. With increased generality and easier programmability, CUDA could be utilized for GPGPU with general mapping principles rather than ad-hoc implementations \cite{11}. In addition, with a similar programming model as the emerging Open Computing Language (OpenCL) \cite{3}, CUDA provides its applications desirable portability on other parallel computing units.

Whereas the aforementioned stereo methods were implemented using traditional GPGPU, this paper exploits the computing power of CUDA. Real-time stereo methods with high matching accuracy on CUDA are rarely reported so far. One recent design is \cite{5}, which uses a semi-global algorithm. The algorithm is intrinsically serial \cite{5} and hence hard to accelerate by parallel computing (only 4\times speedup was attained compared to a CPU implementation). This paper proposes a real-time design of accurate stereo matching on CUDA. A leading local stereo approach is adopted for its high data parallelism. The proposed design attains a speedup factor of 52 compared to the CPU counterpart and is two times faster than \cite{5}. To our knowledge, the proposed design is the first accurate local stereo approach on CUDA with real-time performance.
The contributions of the paper are mainly two-fold. First, as an effective refinement technique, a GPU-oriented bitwise fast voting (BFV) method is proposed to improve matching accuracy at the expense of a minor execution time penalty. Much faster than the histogram-based approach, the proposed BFV can be incorporated into other algorithms. Second, techniques and principles to obtain effective speedup using CUDA are exploited. With fast speed and high accuracy, our CUDA-based design is suitable for practical applications.

The rest of the paper is organized as follows. Section 2 presents the framework of the stereo algorithm. The bitwise fast voting method is introduced in Section 3. Section 4 illustrates the design of stereo matching on CUDA. Section 5 shows the experimental results and Section 6 concludes the paper.

2. Stereo Matching Algorithm

Zhang et al. [15] have developed a cross-based stereo matching algorithm and obtained high matching accuracy. As a local stereo method, it has high data parallelism to exploit and is promising to obtain effective speedup on parallel computing architectures. Our algorithm framework is mainly based on [15], with differences in several algorithm components. This section summarizes the whole algorithm of our design to make the paper self-contained.

The algorithm framework is shown in Fig. 1(a). With the left image $I$ and the right image $I'$, the algorithm computes an optimal disparity for each pixel $p$ in the left image. First, support regions are constructed for each pixel in both images. Then, at each disparity $d$, the matching cost between $p = (x_p, y_p)$ in $I$ and $p' = (x_p - d, y_p)$ in $I'$ is computed. The matching cost is aggregated on the overlapping area $U_d(p)$ between support regions $U(p)$ and $U'(p')$.

Figure 1. (a) Framework of the stereo algorithm. (b) The support regions are constructed using a cross-based approach. (c) Arm lengths are decided using a thresholding method with $h_p^+$ as an example.

2.1 Support Region Construction

Support regions are built from upright crosses [15]. As shown in Fig. 1(b), a cross is defined by a quadruple $\{h_p^0, h_p^+, v_p^-, v_p^+\}$, denoting the left, right, up, and bottom arm length, respectively. The support region $U(p)$ is dynamically constructed by merging the horizontal arms of $q$, which is a supporting pixel in the vertical arms of $p$. Using the cross-based approach, the shape of support regions is represented with compact crosses and the aggregation over a 2D region can be decomposed into two 1D aggregations.

To decide the arm lengths $\{h_p^0, h_p^+, v_p^-, v_p^+\}$, we use a thresholding method based on the intensity difference between the anchor pixel and its neighbors in four directions, with $h_p^+$ as an example in Fig. 1(c). The intensity difference between $p$ and its right neighbors $p_i$ (with a distance of $i$ pixels to $p$) is evaluated as $\phi(p, p_i)$ Eq. (1), increasing $i$ from 2 to a maximum value $L$.

$$
\phi(p, p_i) = \begin{cases} 
0 & \max_{c \in \{R, G, B\}} \left| I_c(p) - I_c(p_i) \right| \leq \tau \\
1 & \text{otherwise},
\end{cases}
$$

where $\tau$ is a thresholding value. The smallest $i$ satisfying $\phi(p, p_{i+1}) \& \phi(p, p_{i+2}) = 1$ is chosen as $h_p^+$.

2.2 Cross-Check & Refinement

To suppress isolated image noise in a computationally efficient way, compared to filtering the input images using a median filter [15].

To speedup the algorithm on GPUs, one key challenge is to accelerate the refinement stage. In the disparity refinement of [15], histograms are built at every pixel, which is difficult to effectively accelerate on GPUs. When implemented on a CPU separately, it will restrain the speed of the whole design besides occupying the CPU resources.

In the following section, we propose a GPU-oriented bitwise fast voting method, greatly accelerating the refinement.

3. Bitwise Fast Voting on GPU

As a statistical method, voting is often used to find an optimal value from a set of given estimates. In our stereo algorithm, it is used to refine the result $d_p^0$ selected by WTA. Since pixels in the same support region $U_p$ are likely from the same image structure, they probably have the same disparity value. Therefore, for each pixel $p$, we use the disparities in $U_p$ to vote an optimal value $d_p^*$. To increase confidence of the refinement, only reliable disparities are used.
for the voting. Reliability of the disparities is determined using a cross-check between left and right disparity maps.

The voting-based refinement is an effective technique to improve matching accuracy, as shown in Fig. 2. First, it acts as a regularisation with a piecewise smoothness prior. In homogeneous regions, the aggregated matching costs at different disparities are usually similar and lack of discriminative power. Therefore, the WTA results tend to be noisy. The regularisation can effectively remove the spurious errors, as shown in the ellipse region Fig. 2. Second, the refinement infers the disparity of occluded areas. A local WTA framework is difficult to handle the occluded regions, because there is no correspondence for the occluded pixels in the other view. The refinement can often attain accurate results for the occluded pixels by propagating the disparity information of their neighbors, as shown in the rectangular region Fig. 2.

Let \( N_p \) be the sample set and \( f^*_p \) is the optimal value to be voted from the set. The traditional voting needs to first build a histogram Eq. (2) and then select a maximum Eq. (3),

\[
\mathcal{H}(i) = \sum_{q \in N_p} \delta[f_q - i] \tag{2}
\]

\[
f^*_p = \arg \max_i \mathcal{H}(i) \tag{3}
\]

where \( \delta[\cdot] \) is an impulse function. In terms of fast implementations on CPUs, since the refinement in this paper needs to vote over non-rectangular support regions, many advanced techniques such as integral histogram [10] are inapplicable or hardly efficient. On the other side, due to the serial and random memory updates, histogram calculation is difficult to effectively accelerate on GPUs [8]. To speedup the voting, this paper proposes a GPU-oriented bitwise fast voting (BFV) method. The key idea is simple and illustrated as follows. In the voting, we only care about the maximum regardless of other values. If the votes of the maximum dominate the statistical set, i.e., the maximum is bigger than the sum of the values at the other bins, we can directly derive the maximum without building a histogram.

Equivalently, this assumes \( \mathcal{H}(f^*_p) > 0.5 \times \sum \mathcal{H}(i) \). Since most pixels in one local support region are likely to have the same disparity value, the assumption is reasonable.

Our method directly derives every bit of \( f^*_p \) from \( N_p \). Using the assumption, the \( l \)th bit \( b_l(f^*_p) \) of \( f^*_p \) can be uniquely determined by the \( l \)th bit of the samples in \( N_p \). Specifically, we sum the \( l \)th bit \( b_l(f_q) \), which could be ‘1’ or ‘0’, of all the samples in \( N_p \) as \( B_l^i(N_p) \) Eq. (4). If \( B_l^i(N_p) \) is bigger than \( 0.5 \times ||N_p|| \), the \( l \)th bit of \( f^*_p \) is ‘1’. Otherwise, the \( l \)th bit of \( f^*_p \) is ‘0’ Eq. (5).

\[
B_l^i(N_p) = \sum_{q \in N_p} b_l(f_q) \tag{4}
\]

\[
b_l(f^*_p) = \begin{cases} 1, & B_l^i(N_p) > 0.5 \times ||N_p|| \\ 0, & \text{otherwise.} \end{cases} \tag{5}
\]

In Fig. 3, an example is used to illustrate the algorithm. ‘\( \times \)’ represents a random value of ‘1’ or ‘0’, indicating that the voting result is fully determined by the samples with the value of \( f^*_p \), regardless of the other samples.

The assumption of \( \mathcal{H}(f^*_p) > 0.5 \times \sum \mathcal{H}(i) \) guarantees that the proposed BFV approach obtains the same results as the traditional histogram-based method. However, even when the assumption is violated, the BFV still has a good chance to get the correct results. This is because \( f_q \) could have the same bitwise value as \( f^*_p \) at some bit, even if \( f_q \) is different from \( f^*_p \). For example, if we replace the first ‘9’ in the \( f_q \) column of Fig. 3 with a random value ‘\( \times \)’ (the assumption is violated), the result will still be correct if one of the four ‘\( \times \)’s at bit 1 is the same as \( b_1(f^*_p) \), with \( l = 0, 1, 2, 3 \) respectively. The effectiveness of the BFV method in improving matching accuracy has been confirmed by the experimental results in Fig. 2.

In view of implementation on GPUs, the proposed BFV has several important advantages compared to the histogram-based voting. First of all, it reduces the mem-
Table 1. Execution time of the histogram-based voting (Hist.) on a CPU and the proposed BFV on a GPU

<table>
<thead>
<tr>
<th></th>
<th>Hist. on a CPU (Core Duo 2.66GHz)</th>
<th>BFV on a GPU (GeForce8800 GTX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of inputs (image resolution)</td>
<td>450 × 375</td>
<td>3.6ms</td>
</tr>
<tr>
<td>Input size (∥Np∥)</td>
<td>32 × 32</td>
<td></td>
</tr>
<tr>
<td>No. of bins (n)</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>Hist. on a CPU</td>
<td>500ms</td>
<td></td>
</tr>
<tr>
<td>BFV on a GPU</td>
<td>3.6ms</td>
<td></td>
</tr>
</tbody>
</table>

ory consumption from n to \(\log_2(n)\), where \(n\) is the number of bins. This advantage is essential to effective computing on CUDA. Since CUDA has limited registers for each thread (around 10 to activate the maximum number of threads) and shared memory for each multiprocessor (e.g., 16KB in GeForce8800 GTX), large memory consumption would greatly reduce the number of active threads [11].

Second, our method allows parallel computing among bits. Third, the voting for a 2D region (Eq. 4) is separable and can be completed by two 1D adding procedures.

Thanks to the advantages, the proposed method is enormously fast on GPUs as shown in Table 1. Running on a GeForce8800 GTX GPU, our BFV only needs 3.6ms to compute 450 × 375 votes, with 32 × 32 input size and 256 bins for each vote. Compared to a histogram-based approach on an Intel Core Duo 2.66GHz CPU, a speedup factor of 140 is achieved by the proposed method. The BFV reduces the time penalty of the disparity refinement to be minor, e.g., 3% of total execution time in matching the Teddy images [1].

4. Stereo Matching on CUDA

4.1. CUDA Parallel Processing Architecture

Compute Unified Device Architecture (CUDA) is provided by Nvidia GPUs for general-purpose parallel computing. CUDA is implemented as a set of Single Instruction Multiple Threads (SIMT) processors. One kernel function is executed by multiple threads simultaneously. With huge data-parallel computing power, CUDA is well-suited to speedup algorithms with high data parallelism.

With a unified programming model, CUDA provides more flexibility to utilize GPU resources. Its programming model mainly consists of a hierarchy of threads and memory [11]. Single threads are organized as thread blocks, facilitating data processing coordinately. A grid consists of multiple thread blocks with the same size. Global memory is off-chip and accessible by all threads. Shared memory is on-chip memory and accessible by the threads from the same block. In addition, each thread has a small number of private registers. Key to the performance on CUDA is to activate massive threads on a large number of cores and hide the memory latency with computation [11].

4.2. Parallelize Stereo Matching on CUDA

The framework of our design on CUDA is shown in Fig. 4. Since the input stereo images are frequently accessed and not modified during the whole algorithm, they are stored in the texture memory which has a hardware-implemented cache scheme. The image-level intermediate data are stored in the global memory. There are mainly five function kernels, i.e., building support windows, cost aggregation in horizontal direction, cost aggregation in vertical direction and WTA, cross-check of left and right disparity maps, and refining the WTA results of the left disparity map, respectively. To reduce the data access to the off-chip global memory, the computation of pixelwise raw matching cost is merged into the kernel of horizontal cost aggregation.

Our framework takes the disparity hypothesis \(d_i \in [d_0, d_1, ..., d_{max}]\) as the outermost loop. For both left and right images, we store the smallest matching cost among \([d_0, d_{i-1}]\) and the corresponding disparity value achieving the smallest cost at each pixel. At \(d_i\), based on the aggregated matching cost, the stored values at \(p\) for the left image and at \(p' = (x_p - d_i, y_p)\) for the right image are updated. At the end of the loop, two disparity maps are obtained and can be used for the cross-check. By this method, the storage to get the WTA results is reduced to four images and independent of the disparity range.

In order to utilize the computing power of CUDA, the algorithm is first partitioned into fine-grained parallel subtasks. For each kernel, outputs are fully independent of each other and parallelism granularity of all the kernels is pointwise. Therefore, each function kernel in Fig. 4 is parallelized according to its output data respectively. When matching images with a resolution of \(W \times H\), we create
4.3. Data Reuse Utilizing Shared Memory

Since access to off-chip memory is slow (200~300 cycles), a primary concern to optimize the performance on CUDA is managing the off-chip memory latency [11]. One intrinsic scheme of CUDA is to activate massive thread warps and hide the memory latency by performing zero-overhead scheduling among them. However, it cannot guarantee the peak performance for many algorithms which either have a small number of threads or are bound on memory bandwidth. An effective technique to alleviate off-chip memory access is data reuse utilizing the fast shared memory that is allocated to each thread block.

In our design, all the kernels except the cross-check have great potential of sharing data among threads in one block. We take the kernel of horizontal aggregation to illustrate our technique of data reuse. As shown in Fig. 6, each thread at \( s \) is to aggregate input data over a horizontal line segment, which is covered by the horizontal arms of \( h_s^- \) and \( h_s^+ \). Typically there is a large overlap between the accessed data of thread \( s \) and its neighboring thread \( t \) as shown in Fig. 6. Therefore, we store the data in shared memory for reuse. For a thread block with \( 16 \times 16 \) threads, an apron of data \([9]\) is loaded as shown in the gray area of Fig. 6. The maximum amount of required data is \( 16 \times (16 + L + L) \) with arm lengths varying from 2 to the maximum arm length \( L \).

Considering both implementation efficiency and matching accuracy, we set \( L \) to 16 and store a \( 16 \times 48 \) block of data in shared memory for each thread block. In terms of implementation efficiency, its first advantage is that data in off-chip memory can be loaded to shared memory by all the threads coordinately, i.e., three \( 16 \times 16 \) data blocks are loaded by the \( 16 \times 16 \) thread block consecutively. In addition, when off-chip data is stored in global memory, all the accesses are coalesced to 16-word reading. The proposed technique is applied to other kernels similarly. For instance, in the kernel of vertical aggregation, a \( 48 \times 16 \) block of data is stored in shared memory.

The technique of data reuse greatly reduces the off-chip memory accesses. To give a concise analysis, we can view the aggregation above as a 1D image convolution. Assuming the average of all arm lengths is \( L \), \( (2 \times L + 1) \times W \times H \) pixels have to be loaded for one \( W \times H \) image without using the technique. In contrast, the amount is reduced to \( 3 \times W \times H \) with the proposed scheme of data reuse. For example, the saving factor amounts to 9 when \( L = 13 \). Observe that, since the arm lengths of different pixels vary arbitrarily, direct accesses from the global memory are hard to be coalesced.

5. Experimental Results and Discussions

We have tested the accuracy and the speed of our design using the Middlebury stereo database [1]. The same parameters are used across all the experiments, i.e., the thresholding value \( \tau = 20 \), the maximum arm length \( L = 16 \), and the truncation value of raw matching cost \( T = 60 \). The program completely runs on a GeForce8800 GTX GPU.

Fig. 7(a)~(d) show the produced disparity maps for four standard stereo image pairs. We observe that high matching

---

The code snippet in the image appears to be a CUDA kernel implementation for stereo matching. The snippet is as follows:

```c
#define BLOCKSIZE 16

__global__ void stereoKernel (arguments) {
    
    ...  //get thread index
    tIdx = blockIdx.x * BLOCKSIZE + threadIdx.x;
    tIdy = blockIdx.y * BLOCKSIZE + threadIdx.y;
    
    //compute output data at the position of [tIdx, tIdy]
    
    ...  //with associated input data
}
```

The code is designed to perform horizontal aggregation over a line segment. The `blockDim` and `gridDim` are initialized to allocate 16x16 thread blocks and 16x48 thread blocks, respectively. The `s` and `t` thread indices are calculated to access the corresponding input data, which are then aggregated to produce the output data.

The figure (Fig. 6) illustrates the data reuse among different threads in one thread block by utilizing shared memory. The figure shows how the shared memory is used to cache the data accessed by multiple threads, thereby reducing the off-chip memory access.

The code snippet is intended to show the implementation flow for the horizontal aggregation kernel, highlighting how data is reused among threads to optimize performance on CUDA.
quality has been obtained in most regions. Fig. 7(f)∼(h) illustrate the disparity maps for the Tsukuba image, reported in state-of-the-art methods with real-time performance. Compare to their results, our proposed design preserves object borders more accurately (see Fig. 7(a)), which is often favored by applications of robotics and multiview rendering. Compared to [6] (see Fig. 7(h)), our design effectively improves the matching accuracy by the proposed BFV-based refinement, producing more accurate results in occluded regions and large homogeneous regions. Table 2 reports the percentage of bad pixels, where a disparity error is bigger than 1, in non-occluded (nocc.) regions, all regions, and depth-discontinuity (disc.) regions respectively. Our design and several other algorithms are reported in ascending order of the average percentage of bad pixels (Ave.). We can see that our method obtains the highest rank among the real-time designs.

Table 3 compares the speed of our design and the referenced work. Our design attains 12fps in matching images with a 475 × 375 resolution and 64 disparities. Using CUDA Visual Profiler, we summarize the portions in the total execution time taken by different algorithm components in Table 4. As expected, the cost aggregation kernels within the disparity loop in Fig. 4 consume most of the execution time. Compared to the recent stereo design on CUDA [5], the proposed design is two times faster. Compared to [13], our design is completely running on GPU and frees the CPU. We attain a speedup factor of 52 compared to the CPU counterpart [15] with almost the same matching accuracy. The large speedup mainly attributes to two-fold contributions. First, the stereo algorithm has been optimized for parallel computing and its data parallelism is fully exploited. Second, the resources of CUDA are properly utilized according to their strength and constraints. Whereas [6] was implemented with the traditional GPGPU approach, our design exploits the computing power of CUDA. The comparison also suggests that, compared to the traditional GPGPU, CUDA maintains the huge computational power while providing more generality and programmability. The increased programmability facilitates the implementation of effective algorithm components. The programming model with more generality also increases the portability of CUDA programs.
Table 3. Speed performance of our design compared with several other stereo methods. (MDE/s = Million Disparity Evaluations/second)

<table>
<thead>
<tr>
<th>Algorithm component</th>
<th>Portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build support windows</td>
<td>4%</td>
</tr>
<tr>
<td>Cost aggregation &amp; WTA</td>
<td>91%</td>
</tr>
<tr>
<td>Cross-check &amp; Refinement</td>
<td>3%</td>
</tr>
<tr>
<td>Median filter &amp; Border handling</td>
<td>1%</td>
</tr>
</tbody>
</table>

800