Optimizing Global Stereo Matching on NVIDIA GPUs

Scott Grauer-Gray
Philadelphia, PA, USA
sgrauerg@gmail.com

***Abstract*—This work presents optimizations to CUDA belief propagation for stereo processing on the GPU to significantly speed up the implementation without hurting the output disparity map accuracy. The optimizations speed up the implementation by over 2x in some cases and are shown to work across a variety of stereo sets and NVIDIA GPU architectures.**

I. Introduction

The retrieval of an accurate disparity map from a set of stereo images is a known computer vision problem. Many methods have been proposed, implemented, and evaluated, often resulting in tradeoffs involving speed and accuracy, where the faster methods generally result in a less accurate disparity map. In particular, local methods that only consider the current or a few neighboring pixels when retrieving disparity are generally faster but less accurate than global methods that take the entire images into account when processing.

One global method that is known to produce relatively accurate disparity maps is belief propagation. Specifically, this method involves the use of Markov random field (MRF) models to generate an NP-hard energy minimization problem for retrieving the disparity at every pixel, and then using belief propagation (BP) to generate an approximate a solution with reasonable computational cost. Sun et. al. [7] introduced this method for stereo matching in 2003. In 2004, Felzenwalb and Huttenlocher [2] introduced three improvements to improve the runtime of belief propagation for stereo matching without changing accuracy. This work is accompanied by C code[1] that implements these improvements and runs on a single thread on the CPU.

During the 2000s, GPUs became more powerful and started being used for general purpose computing on GPUs (GPGPU), as the GPU can generally accelerate applications where processing can be run in parallel on many threads. Belief propagation for stereo matching is an obvious candidate for GPU acceleration since it involves many independent operations on at least half the image pixels in every major step.

In separate works presented in 2006, Brunton et. al. [1] and Yang et. al. [8] ported belief propagation to the GPU using a graphics API with vector and fragment shaders, as specific GPGPU APIs was not yet released at the time of the work. After the release of CUDA for GPGPU in 2007, Grauer-Gray et. al. [4] ported belief propagation for stereo matching to the CUDA environment and presented work in 2008 that showed a significant speedup compared to Felzenwalb's CPU implementation. In 2010, Grauer-Gray and Cavazos [3] presented work showing further optimizations to improve the GPU implementation runtime, specifically optimizing a CUDA kernel in the implementation to use shared memory and registers to store frequently-accessed data rather than slower local memory.

[1]Code available at http://cs.brown.edu/people/pfelzens/bp

II. Belief Propagation For Stereo Matching

As described by Grauer-Gray and Cavazos [3], belief propagation for stereo matching as implemented with the speedups developed by Felzenwalb consists of the following steps:

1) Calculate the data cost for each pixel at each disparity in the disparity space at the bottom level of the computation hierarchy.
2) Iteratively calculate the data costs at each succeeding level of the hierarchy.
3) For each level in the hierarchy (starting from top):
    a) For each pixel in the current checkerboard set, compute the message to send to its four-connected neighbors in the alternate set using the current message values and data cost. Repeat for i iterations, alternating between the two checkerboard sets.
    b) If not at the bottom level of the hierarchy, copy the message values at each pixel to a 2 X 2 block of corresponding pixels in the succeeding level of the hierarchy.
4) Retrieve the disparity estimate at each pixel using the current message values and data costs, with the output corresponding to the disparity that minimizes the sum of the current message values and data cost at the pixel. The disparity estimates across every pixel represent the output disparity map.

The previous work in accelerated belief propagation showed that each step can be parallelized on the GPU, with the result being a significant speedup over the CPU implementation with no decrease of accuracy in the resulting disparity map compared to the ground truth.

III. Stereo Sets

Five stereo sets from the Middlebury Stereo Datasets ([5] and [6]) are benchmarked in the initial and optimized implementations in this work. Specifically, the stereo sets used are the Tsukuba, Venus, Barn1, and Cones (quarter-sized and half-sized). These stereo sets contain a variety of

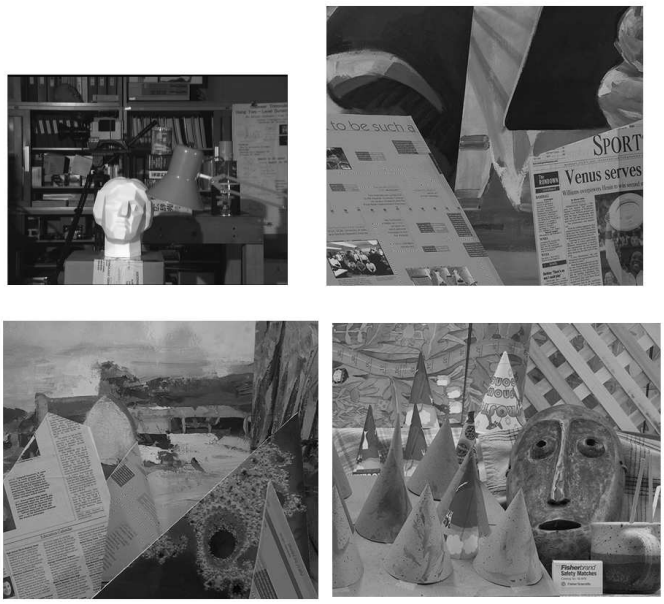

Fig. 1. Reference images from stereo sets benchmarked in this work. Images shown are from (clockwise from upper left) Tsukuba, Venus, Barn1, and Cones stereo sets.

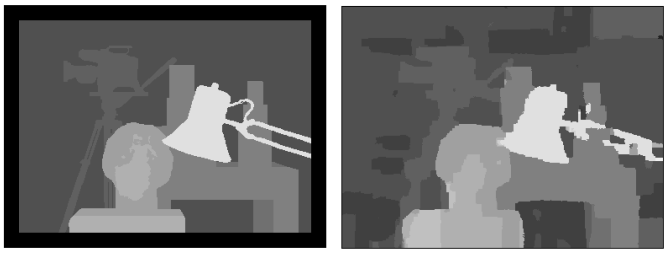

Fig. 2. Ground truth Tsukuba stereo set disparity map (left) and Tsukuba stereo set disparity map generated with initial CUDA implementation (right).

image dimensions and possible disparity value counts, ranging from the 388 X 284 Tsukuba stereo set with 16 possible disparity values to the 900 X 750 cones (half-sized) dataset with 128 possible disparity values. The reference image from each stereo set is in Figure 1 and details for each stereo set is in Table I.

IV. Initial CUDA Implementation

The initial CUDA implementation is based on the implementation described by Grauer-Gray, et. al. in [4], using the code provided with the paper and updating it to work with the current CUDA version. However, the computed disparity maps from this CUDA code did not exactly match the disparity maps computed using Felzenwalb's CPU implementation due to minor differences in processing (the accuracy as compared to the ground truth wasn't affected). For this work, the CUDA code was tweaked to exactly match the CPU processing at every pixel and output the exact same disparity map as the CPU implementation. Both the CPU implementation and this initial CUDA implementation use the 32-bit float data type

Stereo Set	Dimensions	Disparities	CPU time	Init CUDA time
Tsukuba	388 X 284	16	155.3	5.9
Venus	434 X 383	21	317.9	9.4
Barn1	432 X 381	32	507.1	11.4
Cones (q)	450 X 375	64	1077.2	27.7
Cones (h)	900 X 750	128	9055.4	223.5

TABLE I
STEREO SETS BENCHMARKED IN THIS WORK WITH THE CPU IMPLEMENTATION RUNTIME AND INITIAL CUDA IMPLEMENTATION RUNTIME ON V100 (RUNTIME GIVEN IN MILLISECONDS).

to store all the data cost and message-passing data and for computations on this data. The CPU and initial CUDA implementation runtimes on the Tesla V100 GPU are shown in Table I and show a large speedup using the CUDA implementation.

One notable optimization in this initial implementation is the data cost/message value array indexing. The data cost and message value arrays contain data for every (x, y) pixel at every possible disparity, but are implemented as flat, 1D arrays in global memory on the GPU and then indexed with a unique location for each x, y, and disparity combination. A naive method of indexing would be to make the y and x values be the first two dimensions as is typical with pixel indexing within an image, and then add the current disparity as the third dimension, resulting in the following formula (where num_disp = number of values in disparity range and disp = current disparity):

$$index = y * width * num\_disp + x * num\_disp + disp$$

However, this indexing is not ideal for memory access coalesce on the GPU where pixels with the y value and neighboring x values are processed simultaneously. Instead, this implementation uses an indexing method where the y value is first dimension, the current disparity is the second dimension, and the x value is the third dimension, which results in the following formula:

$$index = y * width * num\_disp + width * disp + x$$

This indexing makes it so the data cost and message value data arrays are indexed in a way that the memory access is coalesced when reading from and writing to the global memory on the GPU. The naive method of indexing was implemented and compared to this optimized method, and the results show speedups ranging from 1.48x to 4.05x using the optimized indexing compared to the naive method.

In Grauer-Gray and Cavazo's previous optimization work [3], it is observed that the kernel corresponding the step 3a of the belief propagation algorithm in Section II takes almost 70% of the total runtime, so this 'dominant' kernel is the kernel to initially focus on and evaluate when optimizing the kernels.

V. Test Setup

In this work, all the stereo sets are processed using 5 levels in the computation hierarchy with 7 iterations per level, the relative weight of the data cost compared to discontinuity cost is set to 0.1, and no smoothing is applied to the images

(smoothing sigma set to 0.0). The truncated linear model is used for the data and discontinuity costs, with a maximum value of 15.0 for the data cost and the maximum discontinuity cost set to the number of possible disparity levels for the stereo set divided by 7.5.

The GPUs used in this work are the Tesla V100 (Volta architecture), Tesla P100 (Pascal architecture), RTX 2060 (Turing architecture), and GPU on the NVIDIA Jetson TX1 developer board (Maxwell architecture). The RTX 2060 and Jetson TX1 results are obtained using a local computer/developer board, while Google Cloud is used to run the implementations on the V100 and P100. Ubuntu 18.04 with CUDA 10.1 is used in all tests. The CPU results are obtained locally on a desktop computer with a i7-7700K CPU and 32 GB RAM (with the exception of the Jetson CPU results, which are obtained on the Jetson developer board CPU). The GPU runtimes do not include the time to transfer the input stereo set to the GPU and to transfer the output disparity map back to the host.

The ground truth Tsukuba stereo set disparity map and disparity map generated with the initial CUDA implementation are shown in Figure 2. It is worth noting that this output disparity map differs from the output Tsukuba stereo set disparity map presented in the initial CUDA implementation work [4]; the primary reason is because no smoothing is applied to the stereo set images in this work while the smoothing sigma was set to 1.0 in the initial CUDA implementation work.

VI. Optimizations

A. Memory Management

1) Description: The initial implementation contains using cudaMalloc() to allocate GPU global memory space for two sets of four 'message-passing' arrays containing (level_width / 2) * (level_height) * (num_disparities) data values for each belief propagation level, and then using cudaFree() to free the memory when done with processing at that level. Five levels are used for the processing of each stereo set, so this results in 40 cudaMalloc() and cudaFree() instructions in the implementation. Runtime analysis found that these calls add a non-trivial amount of time to the overall CUDA implementation runtime.

To minimize the number of cudaMalloc()/cudaFree() instructions, the implementation is changed to allocate a single large array in the GPU global memory that contains enough space for the data costs as well as the message-passing arrays for every pixel at every disparity at every level. Then a unique offset from the start of this array is computed to mark the start of each individual data cost and message passing array in the implementation at each level. When five belief propagation levels are used, this decreases the number of cudaMalloc()/cudaFree() instructions for the data cost and message-passing arrays from 40 to 1.

2) Results: The memory management change to minimize the number of cudaMalloc()/cudaFree() call resulted in a significant speedup to the implementation on most of the tested stereo sets, with the results on each GPU shown in Table II. Specifically, the speedup ranged from 1.1 times compared to the initial CUDA implementation on the larger 750 X 900 Cones (half-size) stereo set to over 2 times on the smaller Barn1, Venus, and Tsukuba stereo sets. On the larger stereo sets with a larger disparity range, the GPU memory management time takes a smaller portion of the overall implementation runtime compared to the kernel computation and other processing not affected by this optimization, so this result is expected.

GPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
V100	2.7	2.6	2.5	1.6	1.1
P100	3.6	2.4	2.4	1.5	1.1
RTX 2060	1.8	1.5	1.4	1.1	0.97
Jetson TX1	1.2	1.06	1.03	1.01	1.02

TABLE II
SPEEDUP FOR EACH STEREO SET ON EACH TESTED GPU USING MEMORY MANAGEMENT OPTIMIZATION AS COMPARED TO INITIAL CUDA IMPLEMENTATION.

Interestingly, there was a case on the largest tested stereo set (half-sized cones) on the RTX 2060 where a slight slowdown of about 3% was observed with this change, and the analysis found that the slowdown was potentially caused by the time to allocate the single large array. For this particular stereo set on the RTX 2060, the single allocation of the large array actually took more time than the the multiple allocations/freeing of the smaller arrays.

B. 16-Bit Half Data Type

1) Description: The CUDA Toolkit began supporting the 16-bit half datatype in July 2015 with CUDA 7.5. Specifically, CUDA 7.5 added support for the half and half2 datatypes as will as intrinsics supporting them. In addition, the Tegra X1/2, Tesla P100, and all Volta and Turing GPUs have hardware support for 2x speedup on half2 data operations.

The first way the implementation may benefit from using 16-bit half data is that it halves the number of bytes of data loaded and stored to global memory in each kernel run compared to using 32-bit floats. A switch to 16-bit data may result in a speedup if the current implementation is bottlenecked by global memory load and/or store operations.

The second way the implementation may benefit from 16-bit data is by modifying the kernel processing to take advantage of the 2x speedup on half2 data operations where a single half2 instruction can process two values of packed 16-data data. This change should reduce the overall number of instructions across all the kernel threads since each half2 operation on two data values replaces a separate half or float operation for each of the individual values.

To test the first possible benefit, the data and message passing arrays were changed from 32-bit float storage to 16-bit half storage, causing the load, store, and other operations on the array data to be run using half data instead of float data. Then, the 'dominant' kernel (see Section IV) as well as a couple other kernels were modified to support half2 data operations to test the second possible benefit.

2) Results: The change from using 32-bit float data to using 16-bit half data did result in a significant speedup in most

GPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
V100	1.2	1.3	1.3	1.7	1.2
P100	1.1	1.3	1.1	1.5	1.3
RTX 2060	1.5	1.6	1.3	1.8	1.4
Jetson TX1	1.3	1.4	0.99	1.4	1.4

TABLE III
SPEEDUP FOR EACH STEREO SET ON EACH TESTED GPU USING 16-BIT HALF DATA AS COMPARED TO INITIAL CUDA IMPLEMENTATION USING 32-BIT FLOAT DATA.

of the stereo sets on each GPU, with the speedup generally ranging from 1.2 to 1.4 times over using 32-bit float data. The results for each stereo set on each GPU are shown in Table III. The test setup used for these results were with the memory management optimization applied for both data types.

Investigation of the 'dominant' kernel in each implementation using the NVIDIA Nsight Compute tool shows that memory load/store instructions are a bottleneck when processing the kernel and that using 16-bit half data reduces this bottleneck. Specifically, when processing the Tsukuba stereo set, the Nsight Compute tool shows the same number of global load and store instructions using 32-bit and 16-bit data, but the number of texture to L2 cache requests, L2 to texture returns, texture to SM (streaming multiprocessor) returns, and bytes loaded/stored were all halved when using 16-bit data compared to 32-bit data. In addition, the compute SM utilization increases from 11.5% to 30.9% when going from 32-bit to 16-bit data while the memory utilization remains around 80%, making it clear that memory use is a bottleneck. For the Tsukuba stereo set, using 16-bit data eased the memory use bottleneck and significantly decreased the 'dominant' kernel runtime.

Interestingly, the magnitude of speedup does vary across the stereo sets and GPUs with no clear pattern relating the speedup to stereo set dimensions and/or number of possible disparity values. Investigation on the 'dominant' kernel found that using 16-bit half storage results more registers used, more local memory use (when there is register spillover), and also more overall instructions, all changes that could cause the kernel to run slower. While this optimization does show up speedup in every test, these factors may cause the speedup to be less than initially expected.

It is clear from these results that going from 32-bit to 16-bit data significantly improves the runtime in many cases, but the runtime improvement varies depending on the stereo set and is not a guarantee.

The tests of the second possible optimization, half2 operations, found that using half2 operations did not significantly improve runtime over using 16-bit half data with the default operations in most of the stereo sets and could cause a slowdown of over 25% in some cases. Evaluation of the 'dominant' kernel when processing the Tsukuba stereo set found that using half2 operations did reduce the number of overall instructions, but it also reduced the streaming multiprocessor utilization compared to using the default half operations. This indicates that memory load/store operations are the bottleneck so reducing the instructions does not help the runtime. In addition, the changes required to support the half2 data type made loading the message data from neighboring pixels more complicated; this may be why the runtime increased a little compared to using the default 16-bit data operations. The main exception to this result is the half-sized cones stereo set with a 128-value disparity range where there is a speedup of over 7.5% on the V100, P100, RTX 2060, and Jetson TX1 GPUs when using half2 operations compared to simply using 16-bit half data; in this case the memory load/store is likely less of a relative bottleneck due to the greater amount of computation.

It is worth noting that the change to 16-bit half data did cause the output disparity map to differ a little when compared to the output disparity map using 32-bit float data, but the accuracy did not significantly change when compared to the ground truth disparity map.

Stereo Set	CPU time	Init CUDA time	Opt. CUDA Time
Tsukuba	155.3	5.9	1.9
Venus	317.0	9.4	2.8
Barn1	507.1	11.4	3.6
Cones (q)	1077.2	27.7	10.1
Cones (h)	9055.4	223.5	170.5

TABLE IV
V100 RESULTS: RUNTIME OF THE CPU (USING I7-7700K), INITIAL CUDA, AND OPTIMIZED CUDA IMPLEMENTATIONS ON TESLA V100 GPU FOR EACH STEREO SET (RUNTIME IN MILLISECONDS).

Stereo Set	Jetson CPU time	Init CUDA time	Opt. CUDA Time
Tsukuba	867.3	52.3	36.0
Venus	1914.2	107.8	70.1
Barn1	2845.0	203.5	201.1
Cones (q)	4764.6	720.8	518.2
Cones (h)	38472.7	8969.5	6310.7

TABLE V
JETSON TX1 RESULTS: RUNTIME OF THE JETSON CPU, INITIAL CUDA, AND OPTIMIZED CUDA IMPLEMENTATIONS ON THE JETSON TX1 FOR EACH STEREO SET (RUNTIME IN MILLISECONDS). CPU TIME IS THE RUNTIME ON THE ARM CPU ON THE JETSON TX1.

VII. CONCLUSIONS

The optimized CUDA implementation results on the Tesla V100 and Jetson TX1 are shown for each stereo set in Tables IV and V, with the optimized CUDA implementation including both the memory management and 16-bit half data type optimizations (but not using half2 operations). These results as well as the intermediate results in the previous sections show that the presented memory management and 16-bit half data type optimizations can be effective to improve the runtime of CUDA belief propagation for stereo processing without affecting accuracy, and these optimizations generally work across multiple stereo sets and GPU architectures.

However, the effect of the optimizations can differ depending on the stereo set and GPU architecture, and there can be trade-offs that can potentially make each presented optimization contribute to a slowdown, so the effect of these optimizations should be tested for any desired use case before a wide release. The code used in this work is available at https://github.com/sgrauerg6/cudaBeliefProp and is released under the GNU General Public License.

REFERENCES

[1] A. Brunton, C. Shu, and G. Roth. Belief propagation on the GPU for stereo vision. InProc. 3rd Canadian Conf.Computer and Robot Vision, page 76, 2006.

[2] P. Felzenszwalb and D. Huttenlocher. Efficient beliefpropagation for early vision. InIEEE Int. Conf. Com-puter Vision and Pattern Recognition (CVPR04), pages 261268, 2004.

[3] S. Grauer-Gray, J. Cavazos. Optimizing and Auto-tuning Belief Propagation on the GPU. In The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC) 2010.

[4] S. Grauer-Gray, C. Kambhamettu, K. Palaniappan. GPU Implementation of Belief Propagation Using CUDA for Cloud Tracking and Reconstruction. In 5th IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS) 2008.

[5] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. International Journal of Computer Vision, 47(1/2/3):7-42, April-June 2002.

[6] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003), volume 1, pages 195-202, Madison, WI, June 2003.

[7] Sun, J., Zheng, N.N., Shum, H.Y.: Stereo matching using belief propagation. IEEETrans. Pattern Anal. Mach. Intell. 25(7), 787800 (2003).

[8] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, andD. Nist er. Real-time global stereo matching using hier-archical belief propagation. In British Machine Vision Conf., pages 989998, 2006.