Optimizing Global Stereo Matching on NVIDIA GPUs and CPUs

Scott Grauer-Gray
Philadelphia, PA, United States

ABSTRACT

This work presents optimizations to belief propagation for stereo processing on the GPU and CPU to significantly speed up the implementation without hurting the output disparity map accuracy. The optimizations speed up runtime by over 2x in some cases over an initial parallel implementation and are shown to work across a variety of stereo sets across multiple NVIDIA GPU and CPU architectures. In addition, this work introduces a initial parallel CPU implementation using OpenMP and SIMD instructions that gives a significant speedup over an initial non-parallel implementation and which is then optimized using the presented optimizations.

1 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 trade-offs 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. [9] 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. [10] 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

2 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.

Stereo Set	Dimensions	Disparities	CPU time	Init CUDA time
Tsukuba	388 X 284	16	107.0	5.9
Venus	434 X 383	21	305.1	9.4
Barn1	432 X 381	32	472.5	11.4
Cones (q)	450 X 375	64	1008.1	27.7
Cones (h)	900 X 750	128	8472.7	223.5

Table 1: Stereo sets benchmarked in this work with the CPU implementation runtime (in ms) and initial CUDA implementation runtime (in ms) on V100.

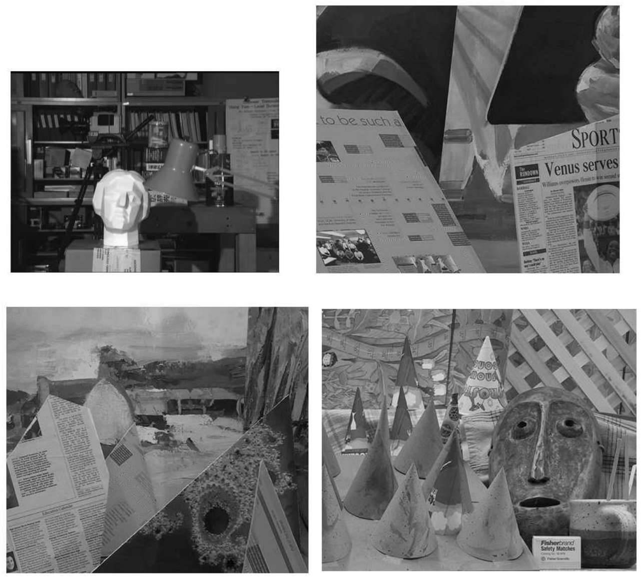

Figure 1: **Reference images from stereo sets benchmarked in this work. Images shown are from (clockwise from upper left) Tsukuba, Venus, Barn1, and Cones stereo sets.**

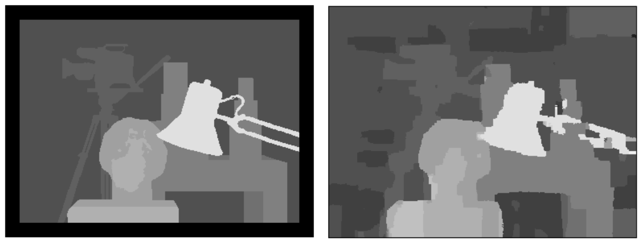

Figure 2: Ground truth Tsukuba stereo set disparity map (left) and Tsukuba stereo set disparity map generated with initial CUDA implementation (right).

3 STEREO SETS

Five stereo sets from the Middlebury Stereo Datasets ([7] and [8]) 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 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 1.

4 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 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 1 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 indices 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 coalescence on the GPU where pixels with the same y index and neighboring x indices are processed simultaneously in the same warp. Instead, this implementation uses an indexing method where the y index is first dimension, the current disparity is the second dimension, and the x index 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; the results are shown in Table 2 and show a significant speedup using optimized indexing across GPUs and stereo sets.

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 2 takes almost 70% of the total runtime, so this 'dominant' kernel is the kernel to initially focus on and evaluate when optimizing the kernels.

GPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
V100	1.5	1.8	2.2	2.1	2.5
K80	2.3	3.0	3.9	3.2	2.9
RTX 2060	2.3	2.6	3.2	2.6	2.1
Jetson TX1	2.1	3.2	2.7	2.0	1.8

Table 2: Speedup for each stereo set on each tested GPU using optimized data cost/message value array indexing compared to naive indexing.

5 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), K80 (Kepler Architecture), RTX 2060 (Turing architecture), and the GPU on the NVIDIA Jetson TX1 Developer Kit (Maxwell architecture). The RTX 2060 and Jetson TX1 results are obtained using a local computer/developer kit, while Amazon Cloud is used to run the implementations on the V100 and K80. Ubuntu 18.04 with CUDA 10.1 is used in all tests except for the Jetson TX1, which uses Ubuntu 16.04 with CUDA 9.0. 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 using the Jetson TX1 Developer Kit CPU). The GPU runtimes include the time to transfer the input stereo set to the GPU and to transfer the output disparity map back to the host. All codes are compiled and linked using the GCC compiler with the -O3 optimization level enabled.

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.

6 CUDA IMPLEMENTATION OPTIMIZATIONS

6.1 Memory Management

6.1.1 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.

GPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
V100	2.9	2.9	2.9	1.6	1.1
K80	1.9	1.9	1.9	1.5	1.2
RTX 2060	1.7	1.4	1.2	1.1	0.97
Jetson TX1	1.3	1.05	1.06	1.03	1.02

Table 3: Speedup for each stereo set on each tested GPU using memory management optimization as compared to initial CUDA implementation.

6.1.2 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 3. 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.

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.

6.2 16-Bit Half Data Type

6.2.1 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 4) as well as a couple other kernels were modified to support half2 data operations to test the second possible benefit.

GPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
V100	1.1	1.3	1.3	1.9	1.4
RTX 2060	1.4	1.7	1.6	1.7	1.8
Jetson TX1	1.05	1.6	1.06	1.6	1.7

Table 4: Speedup for each stereo set on each supported GPU using 16-bit half data as compared to initial CUDA implementation using 32-bit float data. No results for K80 GPU since it does not support 16-bit half data.

6.2.2 Results. The change from using 32-bit float data to using 16-bit half data did result in a significant speedup in most of the stereo sets on each GPU, with the speedup ranging from a low of 1.05 to a high of 1.9 times over using 32-bit float data. The results for each stereo set on each GPU are shown in Table 4. 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 showed 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 showed the same number of global load and store instructions when 16-bit data compared to 32-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 did 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 did show speedup in every test, these factors may cause the speedup to be less than initially expected.

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, 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.

6.3 Data Alignment

6.3.1 Description. Another possible optimization is data alignment, specifically ensuring that the array data corresponding to the start of each row (where x=0) of the stereo set processing is aligned to a specified n-byte boundary. In the first generation of CUDA devices with compute capability of 1.0, global memory accesses that were not aligned to the transaction size (32, 64, or 128-bytes) were very costly and caused many additional transactions with global memory [5]. The penalty is much lower in current CUDA devices, but a performance penalty is still possible if the data is misaligned. For this optimization test, the start of each data array is aligned to a 64-byte boundary and each row at each computation level is padded to a multiple of 16, making the start of each row 64-byte aligned when using float data and 32-byte aligned when using 16-bit half data.

GPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
V100	0.99 / 0.99	1.1 / 1.08	1.01 / 1.00	1.2 / 1.05	1.2 / 1.2
K80	1.1 / N/A	1.1 / N/A	1.07 / N/A	1.06 / N/A	1.1 / N/A
2060	0.99 / 1.0	1.03 / 1.01	1.0 / 1.01	1.06 / 0.93	1.05 / 1.04
TX1	1.02 / 0.82	1.2 / 0.92	1.03 / 1.07	1.03 / 1.01	1.09 / 1.03

Table 5: Speedup for each stereo set on each tested GPU using data alignment when using 32-bit float data/16-bit half data (only float data on K80 since it does not support 16-bit half data).

6.3.2 Results. The data alignment optimization test results are in Table 5 and show that this optimization usually results in a speedup, with the tested speedup topping out at around 20%. The results certainly show that making the data aligned in this manner should be looked at when trying to make the implementation run as fast as possible on an NVIDIA GPU. However, there were a few cases, particularly when using 16-bit data on the smaller stereo sets, where this optimization actually caused a slowdown, so this optimization should not be blindly applied to every scenerio.

6.4 CUDA Optimizations Results

The optimized CUDA implementation results on the Tesla V100 and Jetson TX1 are shown for each stereo set in Tables 6 and 7, with

Stereo Set	CPU time	Init CUDA time	Opt. CUDA Time
Tsukuba	107.0	5.9	1.8
Venus	305.1	9.4	2.4
Barn1	472.5	11.4	3.3
Cones (q)	1008.1	27.7	8.2
Cones (h)	8472.7	223.5	131.1

Table 6: V100 Results: Runtime (in ms) of the CPU (using i7-7700K), initial CUDA, and optimized CUDA implementations on Tesla V100 GPU for each stereo set.

Stereo Set	Jetson CPU time	Init CUDA time	Opt. CUDA Time
Tsukuba	545.6	75.0	54.5
Venus	1696.4	107.8	74.6
Barn1	2724.3	203.5	191.2
Cones (q)	4491.4	720.8	454.5
Cones (h)	36153.3	8969.5	5635.0

Table 7: Jetson TX1 Results: Runtime (in ms) of the Jetson CPU, initial CUDA, and optimized CUDA implementations on the Jetson TX1 for each stereo set. CPU time is the runtime on the ARM CPU on the Jetson TX1.

the optimized CUDA implementation including the memory management optimization, the 16-bit half data type optimization (but not using half2 operations), and using the better result of applying/not applying the data alignment optimization. These results as well as the intermediate results in the previous sections show that the presented memory management, 16-bit half data type, and data alignment 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.

7 OPTIMIZING IMPLEMENTATION ON CPU

The naive, initial, and current optimized CUDA implementations are all significantly faster than the initial CPU implementation. However, that is not a fair comparison between the CPU and GPU, since the initial CPU code is an unoptimized single-thread implementation while the CUDA implementation is a parallel implementation that uses many GPU cores. For example, Lee et. al. [6] looked at GPU codes that were claimed to be 10x-1000x faster on the GPU compared to the CPU and found that when the CPU implementations are optimized, the average GPU speedup dropped to around 2.5x. For a fair comparison, the CPU belief propagation implementation was parallelized and optimized in a manner similar to the CUDA implementation, and the implementation was run and tested on the 4-core / 8-thread Intel i7-7700K, the 24-core / 48-thread Intel Xeon Platinum 8175M CPU, the AMD EPYC 7571 CPU (full CPU has 32 cores, but 24-cores / 48-threads used in this testing), and the 4-core / 4-thread ARM CPU on the Jetson TX1 board.

7.1 Parallel CPU Implementation

The initial CUDA implementation was used as the basis for the parallel CPU implementation. However, instead of using CUDA kernels, each parallelizable portion was made parallel using OpenMP pragmas, with the number of parallel threads set to the number of simultaneous threads allowed each target CPU. Then, the code corresponding to the ‘dominant’ kernel as identified in Section 4 was further optimized using the best possible SIMD instructions available for the architecture. Specifically, AVX2 instructions were used for the i7-7700K and AMD EPYC 7571 CPUs, AVX-512 instructions were used for Intel Xeon Platinum 8175M CPU, and NEON instructions were used for the ARM CPU on the Jetson TX1 board. One important difference between each of these SIMD instructions is that AVX-512 instructions process 16 32-bit float values simultaneously, AVX2 instructions process 8 float values simultaneously, and NEON instructions process 4 float values simultaneously; based on this information the largest expected speedup from SIMD instructions would be from using AVX-512 instructions. The CPU implementations are compiled and linked using the GCC compiler with the -O3 optimization level enabled.

The average speedup of parallelizing the CPU implementation with OpenMP only and with OpenMP + SIMD instructions across the five benchmarked stereo set on each tested CPU is shown in Table 8. Both parallel implementations are significantly faster than the initial CPU implementation on every tested CPU, with the use of SIMD instructions in the ‘dominant’ kernel generally increasing the speedup by 1.5x-2.0x over the OpenMP only parallel implementation. As expected, the parallel implementation speedup is greater when the target CPU has more cores/threads and the SIMD instructions that gave the largest speedup compared to the OpenMP-only implementation are AVX-512 instructions. In addition, the results relative to the i7-7700K CPU show that the implementation has the fastest runtime on the Xeon 8175M CPU compared to the other tested CPUs, with the average runtime 3.1x faster on the Xeon 8175M CPU compared to the i7-7700K.

CPU	OpenMP	OpenMP + SIMD	Speedup vs. i7
i7-7700K	3.2	5.0	1.0
Xeon 8175M	11.0	20.9	3.1
EPYC 7571	12.3	18.9	2.4
Jetson TX1	2.4	3.8	0.15

Table 8: Average speedup across stereo set on each tested CPU using parallelized implementations with OpenMP and with OpenMP + SIMD instructions compared to the initial CPU implementation; rightmost column shows relative speedup of OpenMP + SIMD implementation compared to i7-7700K.

7.2 CPU Implementation Optimizations

To try and further speed up the CPU implementation, each of the CUDA implementation optimizations from Section 6 were applied to the CPU implementation and tested on each CPU.

First, the memory management optimization from Section 6.1 was applied to the CPU implementation with the results shown in Table 9. Interestingly, the result show that this change often caused a slowdown in the CPU runtimes compared to the initial parallel implementation and as a result is not included in the overall optimized implementation on the CPU.

CPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
i7-7700K	0.98	0.99	0.99	0.99	0.99
Xeon 8175M	0.93	0.94	1.00	1.05	1.01
EPYC 7571	0.94	0.93	0.93	1.00	0.93
Jetson TX1	0.62	1.02	1.02	0.97	1.00

Table 9: Speedup (or slowdown) for each stereo set on each tested CPU using memory management optimization.

Then, the optimization using 16-bit floats from Section 6.2 was added to the CPU implementation. This optimization needed to be implemented differently on the CPU compared to CUDA due to the lack of arithmetic operations supporting the 16-bit float data type on the x86/ARM CPUs in the test set; the only operations available are for conversion between 16-bit and 32-bit floats. As a result, 16-bit floats used to store the data cost/message passing arrays in the implementation, but are converted to 32-bit floats for processing and converted back to 16-bit floats when the result is written back to memory. The CPU implementation results using 16-bit floats are shown in Table 10 and show that even with this limitation, there is often a significant speedup using 16-bit floats in the CPU implementation, with the speedup using this optimization ranging from 1.03x to 1.92x.

CPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
i7-7700K	1.92	1.83	1.83	1.76	1.67
Xeon 8175M	1.29	1.72	1.79	1.80	1.71
EPYC 7571	1.03	1.37	1.46	1.40	1.97
Jetson TX1	1.05	1.57	1.06	1.57	1.71

Table 10: Speedup for each stereo set on each tested CPU when using 16-bit 'half' data instead of 32-bit floats.

The final optimization is the data alignment optimization from Section 6.3 where the start of each data array is aligned to 64-byte boundary and each row at each computation level padded to a multiple of 16. This optimization is notable with the SIMD AVX instruction usage on x86 processors since it allows aligned load / store AVX instructions to be used rather than unaligned instructions. The results after adding this optimization to the CPU implementation are shown in Table 11 and show a speedup in most cases, though there are a few notable exceptions where this optimization causes a slowdown; the speedup using this optimization ranges from a slowdown of 0.86x to a speedup of 1.28x.

CPU	Tsukuba	Venus	Barn1	Cones (q)	Cones (h)
7700K	1.03 / 1.05	1.01 / 1.02	1.00 / 1.00	1.2 / 0.99	1.03 / 1.01
8175M	1.25 / 1.28	1.13 / 1.09	1.20 / 1.07	1.21 / 1.11	1.07 / 1.02
7571	1.19 / 1.00	1.02 / 0.99	1.0 / 0.99	1.06 / 0.96	1.05 / 0.86
TX1	1.02 / 0.96	1.09 / 1.11	1.06 / 1.06	1.06 / 1.01	1.08 / 1.10

Table 11: Speedup for each stereo set on each tested CPU using data alignment optimization when using 32-bit float data/16-bit half data.

7.3 CPU Optimizations Results

The 24-core/48-thread Intel Xeon Platinum 8175M CPU gave the fastest CPU runtime on each stereo set of the CPUs tested. The single-threaded, initial parallel, and optimized implementation runtimes on this CPU is shown for each stereo set in Table 12, with the optimized implementation including the 16-bit half data type and the data alignment optimizations, but not the memory management optimization since that often caused a slowdown. As expected, the results show a significant speedup over the initial single-threaded implementation and over the unoptimized parallel implementation.

It is interesting to compare the 8175M CPU implementation results to the CUDA implementation results when run on the Tesla V100, since the Intel Xeon Platinum 8175M CPU and Tesla V100 GPU are both designed to be used in data centers and have somewhat similar power use; the V100 (SXM2 version) has a 250 Watt max, while the Intel Xeon Platinum 8175M CPU power is not listed but the similar Xeon Platinum 8168 CPU has a TDP of 205W (difference is that the 24 cores on the 8168 CPU are clocked at 2.7 GHz as opposed to 2.5 GHz on the 8175M). The comparison shows that the optimized CUDA implementation on the V100 is faster than the optimized CPU implementation on (somewhat) comparable hardware, with the V100 GPU speedup ranging from 1.9x to 4.2x over the 8175M CPU. This result is not surprising given that the GPU is designed to operate on many image pixels in parallel, which is what happens in many steps of the implementations. It is interesting to note that the optimized CPU implementation on the 8175M CPU is significantly faster than the naive CUDA implementation on the V100, showing that optimization is important even when the architecture is ideal for the given algorithm.

Another interesting comparison is between the ARM 4-core CPU and the GPU w/ 256 CUDA cores on the Jetson TX1 development board. The initial and optimized runtimes for the CPU implementation are in Table 13, and the GPU implementation runtimes from the previous section are in Table 7. In this comparison, the GPU again comes out ahead when comparing the optimized implementations, with the GPU speedup ranging from 1.9x to 3.7x compared to the CPU on the Jetson TX1.

8 CONCLUSIONS

The results of this work show that global stereo matching using belief propagation can be significantly sped up on the GPU and CPU with optimized memory management, using 16-bit half data rather than 32-bit float, and with more optimized data alignment. In addition, this work shows that a parallel and optimized CPU implementation is significantly faster than the initial single-threaded

Stereo Set	Init CPU	Parallel CPU (w/ SIMD)	Opt. CPU
Tsukuba	144.9	14.8 (7.1)	3.5
Venus	398.0	34.5 (17.3)	8.9
Barn1	637.5	52.3 (29.5)	13.8
Cones (q)	1355.8	133.9 (73.4)	33.1
Cones (h)	11512.5	1044.5 (598.2)	312.5

Table 12: Xeon Platinum 8175M CPU Results: Runtime (in ms) using initial single-threaded CPU implementation, initial parallel CPU implementation (w/ AVX-512 SIMD instructions), and optimized CPU implementations on 8175M CPU for each stereo set.

Stereo Set	Init CPU	Parallel CPU (w/ SIMD)	Opt. CPU
Tsukuba	545.6	215.9 (134.2)	106.3
Venus	1696.4	610.4 (364.4)	275.7
Barn1	2724.3	916.2 (590.4)	464.5
Cones (q)	4491.4	2373.4 (1439.3)	1160.7
Cones (h)	36153.3	23288.1 (13748.6)	10903.9

Table 13: Tegra TX1 ARM CPU Results: Runtime (in ms) using initial single-threaded CPU implementation, initial parallel CPU implementation (w/ NEON SIMD instructions), and optimized CPU implementations on the TX1 CPU for each stereo set.

implementation, but an optimized CUDA implementation on the flagship Tesla V100 GPU is still faster than using any currently-available CPU. 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. In Proc. 3rd Canadian Conf. Computer and Robot Vision, page 76, 2006.

[2] P. Felzenszwalb and D. Huttenlocher. Efficient belief propagation for early vision. In IEEE Int. Conf. Computer Vision and Pattern Recognition (CVPR 2004), pages 261-268, 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] Harris, M. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. NVIDIA Website, Jan 2013. https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels.

[6] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D.Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, âĂIJDebunking the 100X GPU vs.CPU myth: an evaluation of throughput computing on CPU and GPU,âĂİ in Proc. ISCA, 2010.

[7] 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.

[8] 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.

[9] Sun, J., Zheng, N.N., Shum, H.Y. Stereo matching using belief propagation. IEEETrans. Pattern Anal. Mach. Intell. 25(7), 787-800 (2003).

[10] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nist ÌĄer. Real-time global stereo matching using hier-archical belief propagation. In British Machine Vision Conf., pages 989-998, 2006.