Optimized Parallel Belief Propagation on NVIDIA GPUs and CPUs

Scott Grauer-Gray
Morristown, NJ, United States

Abstract

Parallel processing is a common way to speed up many computer vision algorithms including stereo matching. This work looks at optimized parallel implementations of belief propagation for stereo processing on NVIDIA GPU and x86/ARM CPU architectures and shows runtime comparisons across multiple GPU and CPU processors on a variety of input stereo sets. The work goes on to present and show results of retrieving an optimized parallel configuration for each input stereo set, speedups/slowdowns when using 16-bit floats and 64-bit doubles compared to 32-bit floats, and speedups when using templated disparity counts that allow the iteration counts of loops that iterate through possible disparities to be known at compile time.

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 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. [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.

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

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. [6] 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 [5] 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.

2 Belief Propagation For Stereo Matching

As described by Grauer-Gray and Cavazos [5], 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.

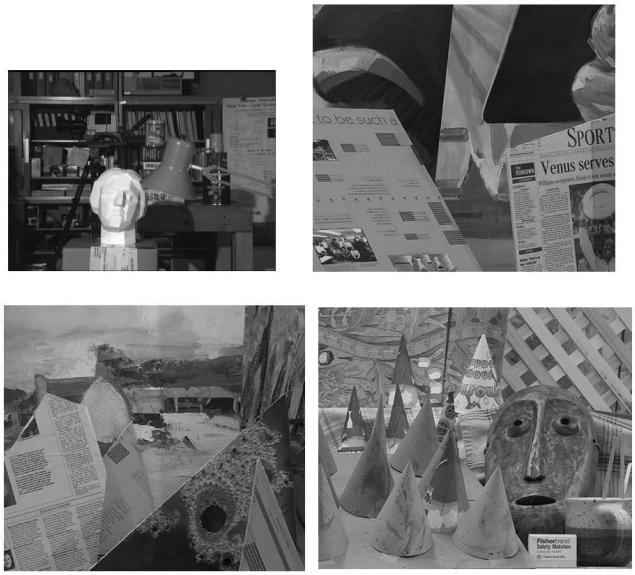

Figure 1. Reference images from stereo sets benchmarked in this work. Images shown are from (clockwise from upper left) Tsukuba, Venus, Barn1, and Cones stereo sets.

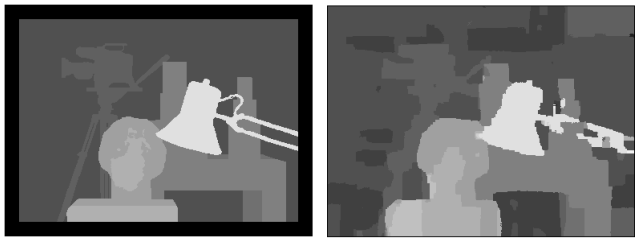

Figure 2. Ground truth Tsukuba stereo set disparity map (left) and Tsukuba stereo set disparity map generated with initial CUDA implementation (right).

Stereo Set	Dimensions	Disparities
Tsukuba (scaled down)	192 X 144	8
Tsukuba	388 X 288	16
Venus	434 X 383	21
Barn1	432 X 381	32
Cones (quarter-size)	450 X 375	64
Cones (half-size)	900 X 750	128
Cones (full-sized cropped)	900 X 750	256

Table 1. Stereo sets from the Middlebury Stereo Datasets that are benchmarked in this work.

3 Stereo Sets

Seven stereo sets are benchmarked in the parallel implementations in this work. Five of the stereo sets are directly from the Middlebury Stereo Datasets ([7] and [8]); these stereo sets are Tsukuba, Venus, Barn1, and Cones (quarter-sized and half-sized). The other benchmarked stereo sets are a scaled down Tsukuba stereo set where the width/height of the images and disparity count are half of the original Tsukuba stereo set and a cropped full-sized Cones stereo set where a 900x750 region around the center is cropped out of the stereo reference and test images. These stereo sets contain a variety of image dimensions and possible disparity value counts, ranging from the 192 X 144 scaled-down Tsukuba stereo set with 8 possible disparity values to the 900 X 750 cones (full-sized cropped) dataset with 256 possible disparity values. The reference images of the Tsukuba, Venus, Barn1, and Cones stereo sets are in Figure 1 and details for each stereo set is in Table 1.

Processor	Architecture	Cores	TDP
H100 (GH200)	NVIDIA GPU	16896	450W-1000W
H100 (PCIe)	NVIDIA GPU	14592	350W
A100 (SXM4)	NVIDIA GPU	6912	400W
RTX 3090 Ti	NVIDIA GPU	10752	450W
AMD Genoa (EPYC 9R14)	x86 CPU	96	400W
Emerald Rapids (Xeon 8559C)	x86 CPU	48	330W
Graviton4	ARM CPU	96	Not Disclosed
Azure Cobalt	ARM CPU	96	Not Disclosed
NVIDIA Grace	ARM CPU	64	Not Disclosed

Table 2. Information about each processor that is benchmarked in this work. Will note that a NVIDIA GPU core is not equivalent to a CPU core, and the TDP for AMD Genoa and Emerald Rapids processors are from online searches and not official (but seem to make sense).

4 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 NVIDIA H100 w/ Hopper architecture and the A100 and RTX 3090 Ti w/ Amphere architecture. The RTX 3090 Ti results are obtained using a local computer, while cloud computing from Lambda Labs is used to run the implementations on the H100 and A100. For the H100 benchmarking, both the GH200 instance as well as an instance with the H100 PCIe are benchmarked, with the H100 in the GH200 instance expected to be faster due to having more CUDA cores enabled and a higher TDP.

The A100 GPU that is benchmarked uses the SXM4 module that has a greater TDP than the PCIe version. 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. Ubuntu 24.04 with CUDA 12 is used when benchmarking the RTX 3090 Ti, while Ubuntu 22.04 with CUDA 12 is used when benchmarking the other GPUs.

The CPUs used in this work are the AMD Genoa (96 cores) and Intel Emerald Rapids (48 cores) x86 server CPUs as well as the Amazon Graviton4 (96 cores), Azure Cobalt (96 cores), and NVIDIA Grace (64 cores[2]) ARM CPUs. Table 2 contains a summary of all the processors benchmarked in this work. Cloud computing from Amazon Web Services is used for benchmarking the AMD Genoa, Emerald Rapids, and Graviton4 CPUs, Microsoft Azure cloud computing is used for benchmarking the Cobalt CPU, and cloud computing from Lambda Labs is used to benchmark the NVIDIA Grace CPU. Ubuntu 24.04 with g++-13 is used in all CPU tests except for the one on NVIDIA Grace, which uses Ubuntu 22.04 with g++-11. The ARM CPUs support one thread per core while the x86 CPUs support two threads per core via Simultaneous Multithreading (SMT) / Hyper-Threading. However, SMT is disabled in the AMD Genoa test environment so only one thread per core is enabled when benchmarking that processor. Two threads per CPU core are supported in the Emerald Rapids test environment.

The speedups shown in this work are relative to a run using the optimized CPU implementation on the AMD Rome Epyc server CPU with 48 cores. All codes are compiled and linked using the g++ compiler with the -O3 optimization level enabled.

The ground truth Tsukuba stereo set disparity map and disparity map generated with the optimized implementations are shown in Figure 2. This output disparity map differs from the output Tsukuba stereo set disparity map presented in the initial CUDA implementation work [6]; 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.

5 Parallel Implementations

The parallel CUDA implementation in this work is based on the implementation described by Grauer-Gray, et. al. in [6] with updates and added optimizations that are described by Grauer-Gray in [4]. The default thread block dimensions for each kernel in the implementation is 32x4.

The parallel CPU implementation in this work is orginally presented in [4] and uses OpenMP [3] as well as single instruction, multiple data (SIMD) instructions for maximum parallelization in the kernels that dominate the runtime. By default, the implementation uses all the concurrent threads that are supported on the CPU in OpenMP parallelization of the processing loops with AVX-512 SIMD instructions on x86 CPUs and NEON SIMD instructions on ARM CPUs. AVX-512 instructions allow for 512 bits to be processed in a single instruction, which corresponds to 16 32-bit float operations processing simultaneously, while NEON instructions allow for 128 bits to be processed in a single instruction. AVX2 is used on x86 CPUs if AVX-512 is not supported and allows for 256 bits to be processed in a single instruction.

The memory management and data alignment optimizations presented in [4] are used in both the CUDA and CPU implementations in this work. Regarding memory management, all the memory needed for processing on the target device is allocated before the belief propagation runs and is not deallocated until all benchmark runs are completed, so the memory allocation and deallocation time is not included in the runtime.

The parallel implementations support input configurations where the disparity count of the stereo set is known at compile time and cases where it is not. In this work, the more general configuration where the disparity count is not templated is benchmarked and used in the overall results up to Section 8, and comparisons between the two configurations are shown and discussed in Section 8.

Both the CUDA and parallel CPU implementations are at https://github.com/sgrauerg6/cudaBeliefProp and are released under the GNU General Public License.

5.1 Default Configuration Results

The results of running the parallel implementation using default parallel configurations are shown in Table 3 with the processors ordered by average relative speedup across all benchmark stereo sets as compared to a baseline parallel run on an AMD Rome CPU w/ 48 cores. As expected, the CUDA implementation when run on the NVIDIA H100 gives the greatest speedup across all the tested processors. However, the parallel CPU implementation results are competitive with the CUDA results, with most of the CPU runs showing greater or similar speedups as compared to the A100 and RTX 3090 Ti GPUs. This shows that a well optimized parallel CPU implementation that uses OpenMP and SIMD instructions might be a viable alternative to a CUDA implementation in some use cases.

6 Parallel Configuration Optimization

The runtime of the parallel belief propagation implementations can be further reduced by finding an optimized parallel configuration for processing each input stereo set, with the parallel configuration corresponding to the thread block dimensions for each kernel in the CUDA implementation and

[2]NVIDIA Grace CPU has 72 cores according to specs, but calls to "lscpu" and std::thread::hardware_concurrency() output 64 cores/threads for CPU in test environment

Processor	Speedup Over Baseline Runs		
	Small Images	Large Images	All
H100 (GH200)	2.12	3.50	2.69
Graviton4	1.87	2.24	2.00
H100 (PCIe)	1.58	2.42	1.92
Azure Cobalt	1.77	2.15	1.92
Emerald Rapids	1.57	2.07	1.79
AMD Genoa	1.32	2.31	1.78
A100 (SXM4)	1.23	1.91	1.49
NVIDIA Grace	1.61	1.34	1.46
RTX 3090 Ti	1.47	1.31	1.31

Table 3. Average speedups over baseline run on each processor with processors ordered from fastest to slowest. Shows average speedup of runs using benchmark stereo sets from Table 1 with smaller images only (first three sets), larger images only (last three sets), and all stereo sets.

Implementation	Parallel Configuration Options
CUDA (thread block dimensions)	**32x4 (default)**, 16x1, 32x1, 32x2, 32x3, 32x5, 32x6, 32x8, 64x1, 64x2, 64x3, 64x4, 128x1, 128x2, 256x1, 32x10, 32x12, 32x14, 32x16, 64x5, 64x6, 64x7, 64x8, 128x3, 128x4, 256x2
Parallel CPU (OpenMP thread count)	**max_cpu_threads (default)**, (3*max_cpu_threads)/4, max_cpu_threads/2, max_cpu_threads/4, max_cpu_threads/8

Table 4. Parallel configuration options on parallel CPU and CUDA implementations with default configuration listed first. In the CPU options, max_cpu_threads corresponds to number of concurrent threads supported via std::thread::hardware_concurrency() function call.

to the OpenMP thread count in the parallel CPU implementation. The default and complete set of parallel configuration options for each implementation are in Table 4.

Testing found that the best results in the CUDA implementation occurred when the parallel configuration for each kernel is optimized separately (allowing for different thread block dimensions for each kernel), while the best results in the CPU implementation occur when the same OpenMP thread count is used across all parallel processing in the implementation. Therefore, this step consists of finding optimized thread block dimensions for each CUDA kernel in the CUDA implementations and an optimized OpenMP thread count to use across all parallel processing in the CPU implementation.

The optimized parallel configuration for each run is found in a brute force manner. First, there are "test runs" where candidate thread blocks dimensions/OpenMP thread counts are benchmarked across every kernel. Then for the CUDA implementation, an optimized parallel configuration is generated where each kernel is set to use the thread block dimensions with the lowest runtime for the specific kernel across the "test runs". For the parallel CPU implementation, the optimized parallel configuration is set to have an OpenMP thread count that corresponds to the thread count in the test run with the lowest total runtime.

The average speedups on each processor when using this parallel configuration optimization compared to using the default parallel configuration across benchmark stereo sets is shown in Table 5 and shows that this optimization gives an average speedup of around 5% across all benchmark runs. Notably, the speedup using this optimization is around or greater than 20% when processing benchmark stereo sets with smaller images on the AMD Genoa and Intel Emerald Rapids processors, so there are cases where parallel configuration optimization provides a significant speedup.

Processor	Optimized Parallel Config Speedup		
	Small Images	Large Images	All
H100 (GH200)	1.02	1.06	1.04
H100 (PCIe)	1.03	1.06	1.05
A100 (SXM4)	1.05	1.07	1.07
RTX 3090 Ti	1.09	1.06	1.08
AMD Genoa	1.31	1.01	1.14
Emerald Rapids	1.19	1.00	1.08
Graviton4	1.08	1.00	1.03
Azure Cobalt	1.07	1.00	1.03
NVIDIA Grace	1.03	0.99	1.01

Table 5. Average speedups using optimized parallel configuration on each processor compared to default parallel configuration. Shows average speedup of runs using benchmark stereo sets from Table 1 with smaller images only (first three sets), larger images only (last three sets), and all stereo sets.

7 Alternative Data Types

So far in this work, the data type used for data storage and processing in the parallel belief propagation implementations has been the 32-bit float data type. However, the parallel implementations also support using the 16-bit float data type, which can speed up processing with loss of precision, and the 64-bit double data type, which can add precision but increases the runtime. This section shows runtime results using these alternative data type as compared to 32-bit floats.

The benchmarking in this section uses the parallel configuration optimization described in Section 6; optimized parallel configurations are retrieved for each processing data type for each input and used in the corresponding benchmark run.

7.1 16-Bit Float Data Type

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, with bfloat16 datatype support later added in the Amphere architecture. The CUDA processing in the implementation when using 16-bit floats is essentially the same as when using 32-bit floats, making it possible to have the datatype be a template parameter to switch the CUDA implementation datatype between 32-bit float, 16-bit float, and 64-bit double. Support for the half2 datatype is more complicated and is not currently supported in the CUDA implementation.

Previous work in [4] found a significant speedup when using 16-bit floats as compared to 32-bit floats in the CUDA belief propagation implementation and states that "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."

On the CPU, most of the current architectures support storage of processing data using 16-bit floats, but the data must be converted to 32-bit floats in order to run SIMD arithmetic operations on the data and then converted back to 16-bit floats for storage when the processing is done. Even with this limitation, using 16-bit floats does result in a significant speedup in the parallel CPU implementations, and the likely reason for this is that using 16-bit data reduces the memory use bottleneck on the CPU in the same way that it does on the GPU. Notably, the Intel Sapphire Rapids and Emerald Rapids server CPUs do support SIMD operations on 16-bit floats with support for AVX512-FP16 extensions, making the conversion to 32-bit floats unnecessary on those processors.

The average speedups on each processor when using 16-bit floats as compared to 32-bit floats across benchmark stereo sets is shown in Table 6 and shows that the speedup using this optimization ranges from around 1.25x to 1.62x on stereo sets with larger images, with lesser speedups on stereo sets with smaller images. The speedups using 16-bit floats are greatest on the x86 CPUs and RTX 3090 Ti of the processors tested, followed by the NVIDIA data center GPUs with ARM CPUs getting the least speedup. The Emerald Rapids results show speedups with and without using AVX512-FP16 extensions and show an additional speedup of around 15% when using AVX512-FP16 extensions.

It is worth noting that when AVX512-FP16 extensions are used, the resulting disparity map on the Cones (full-sized cropped) stereo set input is not as expected due to floating point overflow in the run because of the reduced range of 16-bit floats compared to 32-bit floats (result is as expected for the other stereo set inputs). The same overflow happens when processing that stereo set in the CUDA implementation when the 16-bit "half" datatype is used but can be resolved in that implementation by using the 16-bit bfloat16 datatype that has a greater range (if it is supported on the target GPU...bfloat16 support in CUDA starts in the Amphere architecture).

	Speedup using 16-bit floats		
Processor	Small Images	Large Images	All
H100 (GH200)	1.05	1.38	1.21
H100 (PCIe)	1.01	1.47	1.24
A100 (SXM4)	1.11	1.39	1.26
RTX 3090 Ti	1.21	1.44	1.37
AMD Genoa	1.14	1.62	1.37
Emerald Rapids	1.08	1.47	1.27
Emerald Rapids (AVX512-FP16)	1.27	1.66	1.46
Graviton4	1.00	1.24	1.10
Azure Cobalt	0.99	1.25	1.14
NVIDIA Grace	0.99	1.46	1.19

Table 6. Average speedups using 16-bit floats in optimized belief propagation processing compared to 32-bit floats. Shows average speedup of runs using benchmark stereo sets from Table 1 with smaller images only (first three sets), larger images only (last three sets), and all stereo sets.

7.2 64-Bit Double Data Type

The CUDA and parallel CPU implementations support processing using 64-bit double values in the same manner as floats, with the data type being a template parameter that gives the capability to switch between 32-bit float and 64-bit double processing. Processing using 64-bit doubles gives greater precision at the expense of speed. For belief propagation, the additional precision of 64-bit doubles is not needed in order to give a "good" result, but it is still interesting to benchmark the implementation using the data type to investigate the slowdown of using 64-bit doubles as compared to 32-bit floats when running the implementation on each processor.

The benchmarking results are in Table 7 and show greater slowdown when the implementation is run on stereo sets

with larger images than with smaller images and that the tested processors with the least slowdown when using 64-bit doubles are NVIDIA data center GPUs (H100 and A100).

	Slowdown using 64-bit doubles		
Processor	Small Images	Large Images	All
H100 (GH200)	0.74	0.62	0.67
H100 (PCIe)	0.72	0.59	0.64
A100 (SXM4)	0.72	0.58	0.64
RTX 3090 Ti	0.56	0.61	0.58
AMD Genoa	0.66	0.52	0.59
Emerald Rapids	0.67	0.51	0.59
Graviton4	0.63	0.48	0.55
Azure Cobalt	0.65	0.47	0.58
NVIDIA Grace	0.60	0.47	0.52

Table 7. Average slowdowns using 64-bit doubles in optimized belief propagation processing compared to 32-bit floats, with lower values corresponding to greater slowdowns. Shows average slowdown of runs using benchmark stereo sets from Table 1 with smaller images only (first three sets), larger images only (last three sets), and all stereo sets.

8 Templated Disparity Counts

As noted in Section 5, the CUDA and parallel CPU belief propagation implementations support input configurations where the disparity count of the stereo set is known at compile time and set as a non-type template parameter and cases where the disparity count is not known at compile time and the implementation is compiled to support any possible disparity count. When the disparity count is known at compile time, the loops that iterate through possible disparity values can be better optimized by the compiler using optimizations such as loop unrolling. In this section, both configurations are run and compared across the benchmark stereo sets with the speedup results of using templated disparity counts in Table 8.

The results in the table show significant speedup using templated disparity counts on smaller stereo sets where the number of possible disparities is less than or equal to 21 and lesser speedups on larger stereo sets. This result isn't surprising since most of the runtime in the belief propagation implementation is in loops that go through all or most of the possible disparity values for each pixel. When the disparity count is templated, those loops can be optimized using loop unrolling and other loop optimizations that are possible when the iteration count is known at compile time. However, there are likely diminishing returns from these optimizations as the loop iteration count increases, so it would be expected that the templated disparity count configuration gives a greater speedup on smaller stereo sets with a lower number of possible disparity values.

In addition, the speedups using templated disparity counts are larger on the NVIDIA GPUs runs compared to the parallel CPU runs, as the average speedup using templated disparity counts on NVIDIA GPUs is as high as 1.89x on smaller stereo sets compared to speedups up to 1.49x on ARM CPUs and less than 1.3x on x86 CPUs.

	Templated Disparity Count Speedup		
Processor	Small Images	Large Images	All
H100 (GH200)	1.89	1.52	1.77
H100 (PCIe)	1.79	1.42	1.65
A100 (SXM4)	1.77	1.36	1.64
RTX 3090 Ti	1.85	1.46	1.77
AMD Genoa	1.26	1.03	1.15
Emerald Rapids	1.21	1.02	1.05
Graviton4	1.49	1.02	1.22
Azure Cobalt	1.40	1.03	1.25
NVIDIA Grace	1.48	1.04	1.23

Table 8. Average speedups using templated disparity count on each processor that allows for loop optimizations compared to configuration where it is not. Shows average speedup of runs using benchmark stereo sets from Table 1 with smaller images only (first three sets), larger images only (last three sets), and all stereo sets.

9 Conclusions

The results of this work show that a well-optimized parallel CPU implementation of belief propagation that uses OpenMP and SIMD instructions can be competitive with an optimized CUDA implementation, though the CUDA implementation on the H100 still had the fastest runtimes across all tested processors. The work also shows how the parallel CPU and CUDA implementations can be sped up further by optimizing the parallel configuration for the input stereo set. The work goes on to benchmark the speedup/slowdown of using 16-bit floats or 64-bit doubles as compared to 32-bit floats for belief propagation processing. Finally, the input configuration option of templated disparity counts is presented and benchmarked.

The code used in this work is available at https://github.com/sgrauerg6/cudaBeliefProp and is released under the GNU General Public License, and the full benchmarking results are available in the BeliefProp/ImpResultsBenchmarkingPaper folder of the repository.

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] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. In Computational Science & Engineering, IEEE, pages 46-55, 1998.

[4] S. Grauer-Gray. Optimizing Global Stereo Matching on NVIDIA GPUs and CPUs. https://sgrauerg6.github.io/OptimizingGlobalStereoMatching.pdf.

[5] 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.

[6] 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.

[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] J. Sun, N.N. Zheng, and H.Y. Shum. 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 er. Real-time global stereo matching using hier-archical belief propagation. In British Machine Vision Conf., pages 989-998, 2006.