Optimizing and Auto-tuning Belief Propagation on the GPU

Scott Grauer-Gray and John Cavazos

Computer and Information Sciences,
University of Delaware

Abstract. A CUDA kernel will utilize high-latency local memory for storage when there are not enough registers to hold the required data or if the data is an array that is accessed using a variable index within a loop. However, accesses from local memory take longer than accesses from registers and shared memory, so it is desirable to minimize the use of local memory. This paper contains an analysis of strategies used to reduce the use of local memory in a CUDA implementation of belief propagation for stereo processing. We perform experiments using registers as well as shared memory as alternate locations for data initially placed in local memory, and then develop a hybrid implementation that allows the programmer to store an adjustable amount of data in shared, register, and local memory. We show results of running our optimized implementations on two different stereo sets and across three generations of nVidia GPUs, and introduce an auto-tuning implementation that generates an optimized belief propagation implementation on any input stereo set on any CUDA-capable GPU.

1 Introduction

Belief propagation is a general-purpose iterative algorithm used for inference on problems that utilize Bayesian networks, Markov random fields, and other graphical representations. Applications of the algorithm include free energy estimation in proteins, turbo code decoding, and satisfiability.

Sun [16] introduced belief propagation as applied to the stereo vision problem, and implementations that incorporate the algorithm generally output desirable results according to the Middlebury stereo evaluation [14]. The input to the problem consists of two images of the same scene, a reference image and a test image, with each image displaying the scene from a different perspective along the x-axis. The goal is to accurately retrieve the difference, or disparity, in location along the x-axis of the object shown in each pixel of the reference image to the same scene object in the test image. The disparity space refers to the set of possible disparity values at each pixel. The output is given as a disparity map with a disparity estimate at each pixel, and it is often visualized as a 8-bit grayscale image by multiplying the disparity estimates by an appropriate value to cover the 8-bit range. The results are often used to estimate distance to an

object in the scene, as objects corresponding to pixels of greater disparity are closer than objects of corresponding to pixels of lesser disparity.

For this problem, the belief propagation implementation consists of each node in a 2-D grid computing and sending messages to each of its four-connected neighbors in each iteration, where each message can be viewed as a vector containing a value corresponding to each possible disparity. The algorithm must iterate enough times for the message values to converge in order to obtain an accurate output.

Unfortunately, the algorithm has some shortcomings. Many iterations are necessary for the message values to converge, ample storage is needed to hold the message value vectors at each pixel, and the naive computation time of each message vector is of order $O(n^2)$, where n refers to the number of values in the disparity space. Some applications of stereo vision require real-time processing, and a naive belief propagation implementation on the CPU suffers from slow run-time and high storage requirements.

Felzenszwalb [6] presents methods to mitigate these downsides, presenting a hierarchical scheme to reduce the number of iterations necessary for message value convergence, a checkerboard scheme that reduces storage requirements by allowing updates to be performed 'in place' and that halves the number of messages computed and sent in each iteration, and an algorithm that generates the message vectors in $O(n)$ time using certain discontinuity models. These methods are regularly applied in belief propagation implementations for stereo processing.

The running time of a belief propagation implementation can be reduced further by taking advantage of the parallel processing capabilities of the graphics processing unit (GPU), as each step of the algorithm can be performed in parallel on each pixel. Brunton [2], Yang [18], Grauer-Gray ([8] and [7]), Xu [19], Liang [10], and Ivanchenko [17] present implementations of belief propagation for stereo processing on the GPU. Brunton and Yang map their implementations to the graphics API, while Grauer-Gray, Xu, Liang, and Ivanchenko take advantage of the CUDA architecture.

There are challenges to optimizing a program on the CUDA architecture, as noted by Ryoo [13] and Datta [5]. An optimization that decreases the running time of one program may increase the running time of another program, or of the same program with a change of parameters, and the impact of an optimization may vary across GPUs with non-uniform architectures.

In this paper, we focus on optimizations that can be applied to a CUDA implementation of belief propagation as applied to stereo vision. We explore ways to minimize the number of accesses to high-latency local memory on the GPU. Data in local memory is frequently accessed in our initial implementation, and a decrease in the number of local memory accesses is likely to lead to a faster runtime. We present three optimized CUDA implementations of belief propagation, and go on to present an auto-tuning implementation that can optimize CUDA belief propagation across different stereo sets and GPUs.

```
float m[N];

float currMin = INFINITY;

for (int i=0; i < N; i++)
{
    m[i] = dataC[INDX_D_i] +
         neigh1[INDX_N1_i] +
         neigh2[INDX_N2_i] +
         neigh3[INDX_N3_i];

    if (m[i] < currMin)
        currMin = m[i];
}
.
.
```

(a) Naive Implementation Using Local Memory

```
float m[N];

float currMin = INFINITY;

#pragma unroll
for (int i=0; i < N; i++)
{
    m[i] = dataC[INDX_D_i] +
         neigh1[INDX_N1_i] +
         neigh2[INDX_N2_i] +
         neigh3[INDX_N3_i];

    if (m[i] < currMin)
        currMin = m[i];
}
.
.
```

(b) Optimized Implementation Using Register Memory

```
__shared__ float m_shared[N*THREAD_BLK_SIZE];

float currMin = INFINITY;

for (int i=0; i < N; i++)
{
    m_shared[i_currThread] =
        dataC[INDX_D_i] +
        neigh1[INDX_N1_i] +
        neigh2[INDX_N2_i] +
        neigh3[INDX_N3_i];

    if (m_shared[i_currThread] < currMin)
        currMin = m_shared[i_currThread];
}
.
.
```

(c) Optimized Implementation Using Shared Memory

Fig. 1. Code analogous to a portion of the dominant kernel of our CUDA belief propagation implementation; shows the naive and optimized register / shared memory implementations.

2 Optimization Overview

As described in the programming guide [1], CUDA allows the GPU to be utilized as a co-processor to the CPU for general-purpose programming, processing a kernel function on multiple threads simultaneously. Each thread contains a unique ID within a 1D, 2D, or 3D thread block structure, and each thread block is part of a 1D or 2D grid structure of thread blocks. Each thread within a block is executed on the same multiprocessor and is processed as part of a 32-thread chunk known as a warp. The number of active warps per multiprocessor is bounded by a GPU-specific maximum, placing a ceiling on the parallelism in the program execution. However, the number of active warps is often limited by the number of registers or the shared memory available on each multiprocessor. The ratio of the actual number of active warps to the maximum number of active warps during the kernel execution represents the multiprocessor occupancy.

To illustrate potential optimizations, we describe the following algorithm which is analogous to a portion of the most heavily-used/dominant kernel of our belief propagation implementation:

1. Set a float variable 'currMin' to infinity.
2. For i = 1 to some N do the following:
   (a) Compute m_i, the sum of the values accessed from specific indices in four separate arrays (the data cost array and the message arrays from three neighbors, using the indices INDX_D_i, and INDX_N1_i, INDX_N2_i, and INDX_N3_i, respectively) in global memory, and store the value to a specific form of storage (local, shared, or register memory).
   (b) Check if m_i is less than the value of 'currMin'; if so, set 'currMin' to m_i.

The set of m_i values are accessed, manipulated, and compared with each other in future operations, so it makes sense to store them in a structure such as an array that allows for easy access via index value.

Our initial implementation causes the m_i values to be stored in an array which is local to the thread, resulting in the utilization of local memory on the GPU. Unfortunately, accesses to data in local memory are slow compared to accesses to data in registers or shared memory. We go on to generate optimized implementations where local memory accesses are converted to register or shared memory accesses. Code corresponding to each implementation is displayed in Figure 1. In the register memory implementation, the loop is completely unrolled via the '#pragma unroll' directive; this changes the implementation such that the array is no longer indexed via a variable value within a loop and is no longer automatically stored in local memory. In the shared memory implementation, the array m_shared is shared across every thread in a thread block of size THREAD_BLK_SIZE; the index i_currThread in the code is unique to each thread in the block; it corresponds to the location of the m_i value stored in the array m_shared for the thread.

Intuitively, one would expect the optimized implementations to be faster than the initial implementation since accesses to high-latency local memory in the initial implementation are replaced with accesses to low-latency registers or shared memory in the optimized implementations. However, the utilization of a greater number of registers per thread in the register implementation and of shared memory in the shared memory implementation limits the number of thread warps which can be processed in parallel, decreasing the multiprocessor occupancy and possibly adversely affecting the running time. The only way to truly determine the effect of these optimizations is to perform experiments which involve comparing the results of the different implementations; we perform such experiments using these optimizations and describe the results in Section 5.

3 CUDA Belief Propagation

In this section, we present a initial CUDA implementation of belief propagation; the implementation utilizes the speed-ups described by Felzenszwalb [6] and 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 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.

Grauer-Gray [8] showed that each of the steps of the algorithm can be performed in parallel using the CUDA architecture, and the resulting disparity map is obtained more quickly using a CUDA implementation as compared to a sequential CPU implementation. However, that work does not discuss optimizations which can be applied to decrease the running time of the CUDA implementation. In this paper, we present optimizations that can be utilized to reduce the running time of a CUDA implementation without affecting the output disparity map.

In our experiments, we first smooth the images using a Gaussian filter of sigma value 1.0, then run belief propagation using 5 hierarchical levels, 10 BP iterations per level, and a disparity space ranging from 0 to 14 in increments of 1. The truncated linear model is used for the data and discontinuity costs, with a maximum value of 15.0 and 1.7 for the data and discontinuity cost, respectively. The relative weight of the data cost as compared to the discontinuity cost is held at .07.

Our initial CUDA experiments utilize the nVidia GTX 285 GPU and are compiled using CUDA toolkit 3.1. The GTX 285 contains 30 multiprocessors, each containing 8 processors for a total of 240 parallel processors, with 16392 registers and 16 KB shared memory available in each multiprocessor. The thread block dimensions are set to 32 X 4, with 32 corresponding to the width and 4 corresponding to the height.

We begin with a naive CUDA belief propagation implementation, using a separate kernel for each step/sub-step of the algorithm, and run our implementation on the 384 X 288 Tsukuba stereo set shown in Figure 2. This implementation runs in 47.0 ms; the resulting disparity map is shown to the right of the stereo set in the figure.

Fig. 2. Left/Middle: Images of Tsukuba stereo set; Right: Computed disparity map using our implementation

We utilize the CUDA profiler to analyze the running time of each kernel and discover that almost 70% of the running time is spent in the kernel which computes four arrays of message values, each to be received by one of the current pixel's four-connected neighbors. As a result, we focus our optimizations on this kernel, which corresponds to step 3a of the aforementioned belief propagation algorithm. Each array of message values is computed in the kernel using the following O(n) algorithm introduced by Felzenszwalb [6]:

1. For each disparity d in the disparity space, initialize the message value m_d to the sum of the data cost and the current message value of each non-recipient neighbor, where the data costs and message values are retrieved from global memory, and retrieve the minimum m_d value; this step corresponds to the code described in Section 2.
2. Set m_{max} to the sum of the minimum m_d value and T_{data}, the truncation value that corresponds to the maximum possible discontinuity cost.
3. Loop from d = 1 to d = max_{disp}, setting each $m_d = min(m_{d-1} + 1, m_d)$, assuming that the values in the disparity space differ by 1.
4. Loop from d = max_{disp-1} to d = 0, setting each $m_d = min(m_{d+1} + 1, m_d)$, assuming that the values in the disparity space differ by 1.
5. Loop from d = 0 to d = max_{disp}, setting each $m_d = min(m_d, m_{max})$, and compute the summation of the output m_d values.
6. Retrieve the average message value by dividing the summation of the output m_d values by the number of message values
7. Loop from d = 0 to d = max_{disp}, setting $m_d = m_d$ - (average message value).
8. Store the resulting message values m_d for each disparity d in the appropriate location in global memory for use in the following iteration or final disparity estimation.

Inspection of the resulting PTX code and the profiling output reveals that local memory is utilized to hold the array that contains the message values that are currently being computed, causing a large number of accesses to the high-latency storage.

In the following sections, we discuss strategies to reduce or eliminate the use of local memory in this array, looking at ways to utilize registers and shared memory rather than local memory.

The number of local loads/stores given in the results correspond to totals across all invocations of this kernel, while the number of registers and the occupancy refers to the resource use in a single invocation of the kernel.

4 Experimental Methodology

We first run experiments using optimized implementations that utilize either registers or shared memory in the array that contains the message values that are currently being computed, go on to introduce a hybrid implementation that combines the usage of register, shared, and local memory in the array, and finally develop an auto-tuning implementation to generate the optimal parameters for the hybrid implementation that works across different stereo sets and GPUs.

We initially perform our experiments using the GTX 285 GPU, and then go on perform some of the same experiments using the Tesla C870 and the GTX 470, CUDA-capable GPUs with architectures that differ from the GTX 285.

5 Optimization Results: Register and Shared Memory Implementations

In this section, we describe the results of applying optimizations to the most heavily used kernel of our CUDA belief propagation implementation; these optimizations are intended to eliminate the use of local memory in the computation of the message values corresponding to each disparity, placing the data in registers or shared memory rather than local memory.

The first optimized implementation utilizes registers to store message values via the method shown in the sample code in Figure 1; this can be viewed as an application of the register promotion/scalar replacement optimization described by Cooper [4] and Callahan [3], where a value in memory is placed in a register for quick access. However, this optimization increases register pressure, which decreases the number of thread blocks which can run in parallel and may cause register spilling into local memory.

The second optimized implementation takes advantage of the shared memory present on each multiprocessor to store message values via the method shown in the right of Figure 1; the utilization of shared memory as a user-managed cache is a common CUDA optimization as described in the programming guide [1]. It is often profitable to load values from global memory into shared memory and then access/update the values from there to take advantage of the low latency associated with shared memory.

The results of running these optimizations are displayed in Table 1. The multiprocessor occupancies did decrease due to increased use of registers/shared memory in the optimized implementations. Still, the total running time on the GTX 285 decreased from 47.0 ms using the initial CUDA implementation to 24.3 ms using the register implementation and to 25.4 ms using the shared memory implementation, corresponding to speed-ups of 1.93 and 1.85, respectively, over the initial CUDA implementation.

Storage/Loop Unroll Setting	Num local loads	Num local stores	Num regs	Occup.	Total running time	Speedup from init. CUDA imp.
Initial CUDA imp. on GTX 285	1307529	7360296	37	0.375	47.0 ms	—
Register memory imp. on GTX 285	0	0	110	0.125	24.3 ms	1.93
Shared memory imp. on GTX 285	0	0	53	0.25	25.4 ms	1.85

Table 1. Running times and resource use of the initial and the optimized register and shared memory CUDA belief propagation implementations on Tsukuba stereo set on the GTX 285. The resource use data corresponds to the most heavily used kernel described in Section 3.

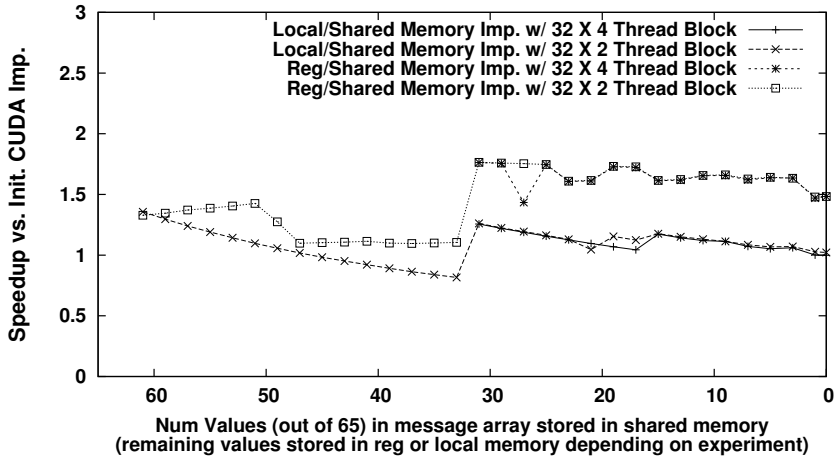

Fig. 3. Speedup of running the hybrid CUDA imp. (vs the initial CUDA imp.) on the 'Cones' stereo set with the GTX 285 using varying amounts of shared and register/local memory.

6 Hybrid Implementation: Multiple Memory Modes In a Single Implementation

Next, we create a hybrid implementation which can utilize shared, register, and local memory in the array used for computation of the message values on the GPU. This allows more values to be placed in low-latency shared/register memory without resorting to high-latency local memory and allows the programmer to direct storage to local memory without dealing with the unpredictable effects of register spilling.

Our implementation allows the programmer to store x values of the array in register memory, y values in shared memory, and z values in local memory. The values in register memory may be spilled into local memory if there are not enough registers allocated to hold them or if any loops accessing them becomes too large to be unrolled, while the number of values which can be stored in shared memory is limited by the shared memory available on the multiprocessor.

We run our hybrid implementation on the GTX 285 using the quarter-sized version of the 'Cones' stereo set included as part of the 2003 Middlebury stereo sets [15]. These images measure 450 by 375 and have a disparity range from 0 to 64; the remaining parameters are the same when processing this image set as the Tsukuba set described in Section 3.

Our initial CUDA implementation described in Section 3 processes the images in 420 ms on the GTX 285; we use this result as the basis for the speed-up of

our optimizations. The optimized register memory implementation described in Section 5 processes the stereo set in 300 ms, while the size of the input disparity range precludes running our optimized shared memory implementation (also described in Section 5); there is not enough shared memory available on the multiprocessor to store the entire array used for message value computation when using a disparity increment of 1 and thread block dimensions of 32 X 4.

We go on to perform four sets of experiments using the our hybrid implementation; the results of each set of experiments on the GTX 285 in terms of speed-up over the initial CUDA implementation are shown in Figure 3, while the data corresponding the best-performing configuration is displayed in Table 2.

In the first set of experiments, we hold the thread block dimensions constant at 32 X 4 and adjust the register/shared memory usage when placing the values in the array used for the computation of message values. In the second set of experiments, we set the thread block dimensions to 32 X 2 to allow greater use of shared memory when updating the message values; these dimensions allow up to 61 message values in each thread to be stored in the 16 KB shared memory available on each multiprocessor on the GTX 285 compared to 31 message values when using 32 X 4 thread blocks. In the third and fourth sets of experiments, we use the same 32 X 4 and 32 X 2 thread block dimensions and utilize shared and local memory rather than shared and register memory.

The best-performing configuration on the GTX 285 gives a speedup of 1.76 times over the initial CUDA implementation; this shows that with the right choice of parameters, our hybrid implementation can be utilized to reduce the running time of CUDA belief propagation.

	Optimal Configuration and run-time data/results							
GPU (config.)	Thread Block Dims	Num vals in reg.-shared-loc. mem. in mess. array	Num local loads	Num local stores	Num regs used	Occup.	Total run-time	Speedup over Init. CUDA imp.
GTX 285	32 X 2	34-31-0	2209184	4418368	124	0.125	238.0 ms	1.76
Tesla C870	32 X 2	14-51-0	0	0	120	0.083	840.2 ms	1.43
GTX 470 (fav L1 cache)	32 X 2	0-7-58	4523184	4339010	61	0.333	231.2 ms	0.94
GTX 470 (fav shared mem)	32 X 2	0-63-2	87327	133654	59	0.125	189.6 ms	1.19

Table 2. Optimal configuration and corresponding results obtained when running our CUDA hybrid implementation on the 'Cones' stereo set using the GTX 285, Tesla C870, and GTX 470 GPUs. The resource use data corresponds to the most heavily used kernel described in Section 3.

6.1 Hybrid Results Discussion

While investigating the results of the hybrid implementation experiments, we discovered that the running time often decreases when moving to a configuration which allows for an increase in occupancy from the previous configuration, and then the running time increases with increased usage of local memory until the parameter set allows for another increase in multiprocessor occupancy.

Based on this observation, we believe that it is possible to run experiments using a single well-generated configuration at each occupancy to retrieve the optimal configuration, rather than searching the entire optimization space. In the next section, we introduce such a auto-tuning system that uses this observation to optimize belief propagation on any CUDA-capable GPU.

7 Auto-Tuning Implementation

In this section, we present our auto-tuning implementation introduced in Section 6.1. At each occupancy, we produce the configuration that maximizes the usage of shared/register memory, with the goal to minimize the number of accesses to local memory. Then, we compare the results across occupancies to retrieve the optimal configuration.

The steps of our auto-tuning implementation are as follows:

1. For the input max occupancy, determine the thread block dimensions. The width is set at 32, while the height is retrieved as follows:
   (a) Retrieve the whole number of rows of length 32 which are to be processed in parallel on each multiprocessor using the input max occupancy and the GPU-specific max number of threads which can be processed in parallel on each multiprocessor.
   (b) Calculate the maximum thread block height (1-16) that allows for this number of rows to be processed concurrently on each multiprocessor, operating under the GPU-imposed constraint that no more than 8 thread blocks can be processed concurrently on each multiprocessor.
   (c) If no thread block height meets the above criteria, decrement the number of rows by one and return to the previous step. This process continues until the criteria is met.
2. Determine the number of registers which can be allocated to each thread based on the thread dimensions, max occupancy, and the number of registers available on each multiprocessor.
3. Set a NUM_REG_VALUES_INITIAL number of values to be stored in register memory in the array used for the computation of message values (note that this data may be spilled to local memory).
4. If there are still values left to be stored in the array, determine the number of values that can be stored in shared memory with the given occupancy, and set the minimum of that value and the number of values that still need to be stored in shared memory.

5. If there are still values to be stored, place the remaining values in register memory until MAX_REG_VALUES are placed in register memory. Then place the remaining values in local memory.

This implementation can be viewed as an application of tiling, as we perform experiments using data partitions of varying sizes via changing the maximum occupancy. A greater portion of the data can be stored in low-latency register and shared memory with a lower multiprocessor occupancy, but at the cost of less parallelism.

In our experiments, we set the NUM_REG_VALUES_INITIAL to 12 and MAX_REG_VALUES to 50 data elements, which leads to register spillover but which led to a faster run-time than specifically placing more of the data in local memory. Then, we test our implementation at each occupancy from 0.04 to 1.00 in increments of 0.02. The results are given in terms of speed-up over the initial CUDA implementation in Figure 4, with the maximum speed-up and the difference from the optimal implementation in Section 6 shown in Table 3. The maximum speedup over the initial CUDA implementation on the GTX 285 using this implementation is 1.76, which is the same as the speed-up found in Section 6 and is generated using fewer trials. In the future, we plan to look into improvements to the framework in order to generate better results in fewer trials.

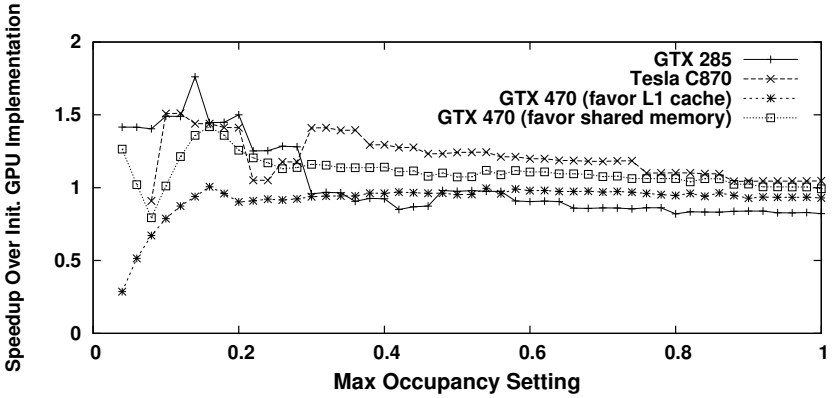

Fig. 4. Results of auto-tuning at different occupancy levels.

8 Experiments Using Different GPUs

To compare the results across multiple generations of GPUs, we perform the hybrid implementation and auto-tuning experiments on the Tesla C870 GPU,

	Optimal Configuration from auto-tuning			
GPU (config.)	Max occupancy	Running Time (ms)	Speed-up over Init. CUDA imp.	% Difference in speedup from opt. imp. from Section 6
GTX 285	0.14	238.3 ms	1.76	0.0%
Tesla C870	0.10	797.7 ms	1.51	+5.06%
GTX 470 (favor L1 cache)	0.58	227.6 ms	0.95	+1.56%
GTX 470 (favor shared mem)	0.16	188.0 ms	1.20	+1.01%

Table 3. Optimal Results obtained from auto-tuning at varying maximum occupancies.

which uses a GPU architecture which proceeded the GTX 285, and the GTX 470 GPU, which utilizes the Fermi architecture that succeeded the GTX 285. The results are shown with the GTX 285 results in Figure 4 and Table 2.

The Tesla C870 GPU utilizes the G80 architecture that proceeded the GTX 285; it contains 16 multiprocessors, each with 8 processors for a total of 128 processors [1]. Each multiprocessor contains 16 KB shared memory and 8192 registers, representing half the number of registers per multiprocessor compared to the GTX 285. Meanwhile, the GTX 470 utilizes the GF100 (Fermi) architecture that succeeded the GTX 285; this GPU contains 14 multiprocessors with 32 processors each for a total of 448 processors. Each multiprocessor contains 32768 registers and 64 KB which is shared between shared memory and L1 cache. The L1 cache on each multiprocessor along with the global L2 cache reduces the latency associated with local memory. The programmer can either allocate 16 KB shared memory / 48 KB L1 cache per multiprocessor or 48 KB shared memory / 16 KB L1 cache per multiprocessor [1]; we perform experiments using each configuration. Our experiments on the GTX 285 and Tesla C870 are compiled and run using CUDA 3.1, while the experiments using the GTX 470 utilize a beta version of CUDA 3.2.

Our experiments using the GTX 470 reveal that unrolling the loops to place the data in registers does not decrease the running time as it does on the GTX 285 and the Tesla C870; one possible reason for this is the L1 cache present on the GTX 470, as this decreases the latency associated with local memory. As a result, we set NUM_REG_VALUES_INITIAL and MAX_REG_VALUES to 0 when performing auto-tuning on this GPU; this places as much data as possible in shared memory given the occupancy, and then places the remaining data in local memory.

The results show that our implementations are flexible across GPUs; our system is able to generate an optimal configuration for each architecture. Interestingly, the optimal implementation retrieved on the GTX 470 when favoring L1 cache does not give a speed-up over the initial implementation; this is likely because the presence of the larger L1 cache results in lower-latency accesses to local memory. Nevertheless, our framework is able to obtain a significantly

faster implementation on the GTX 470 when favoring shared memory; we obtain a speedup of 1.20 times over the initial CUDA implementation using this option.

9 Splitting up the image

Now, we modify the implementation to allow for image splitting in order to increase the flexibility of our implementation across GPUs with varying amounts of DRAM and to relax the constraint on image size and the number of disparity levels. This implementation splits the input images into multiple partitions, runs belief propagation on each partition, and then combines the results.

To prevent inaccurate measurements on the edge of each partition, our implementation allows padding to be applied on each image partition, making the partition size larger than the section included in the output disparity map.

We perform our experiments using the half-sized and full-sized images of the 'Cones' stereo set described in Section 6; these stereo sets measure 900 by 750 with a disparity range from 0 to 128 and 1800 by 1500 with a disparity range from 0 to 255, respectively. We set the padding to 20 pixels in each experiment. On the half-sized 'Cones' stereo set, we divide the image into three rows, and on the full-sized set, we divide the image into 25 partitions (5 ways vertical and 5 ways horizontal). The remaining parameters remain the same as in the previous experiments.

We benchmark our implementations on the GPU using the initial CUDA implementation as described in Section 3. Then, we utilize our auto-tuning implementation to retrieve an optimal configuration on each GPU; the resulting running time and speed-up over the initial CUDA implementation on the image sets using this optimal configuration are shown in Table 4.

As the number of disparity values increases, a greater portion of the data is placed into local memory due to the limited amount of registers and shared memory on each processor. As a result, the optimized results are very similar to the initial CUDA results in these experiments. Future work includes research into methods intended to optimize the running time when there is a larger disparity space.

	Optimal Configuration from auto-tuning		
GPU (config.)	Max occupancy	Running Time	Speed-up over initial CUDA imp.
GTX 285	0.14 (0.14)	3120 ms (31500 ms)	1.18 (0.97)
Tesla C870	0.36 (0.36)	8260 ms (74900 ms)	1.01 (0.94)
GTX 470 (favor L1 cache)	0.54 (0.64)	1980 ms (18600 ms)	0.98 (0.98)
GTX 470 (favor shared mem)	0.58 (0.54)	1940 ms (18500 ms)	1.02 (0.99)

Table 4. Optimal Results on half-sized (full-sized) 'Cones' stereo set obtained from auto-tuning.

10 Related Work

We discussed related work in GPU belief propagation in Section 3. In addition, there is a body of work related to optimizing/auto-tuning on the GPU. Ryoo [13] looked at optimizations targeted to hide the stalling associated with long-latency operations, methods to be distribute the workload, reducing the number of dynamic instructions, and maximizing intra-thread parallelism and resource use on the GTX 8800 GPU. Datta [5] looked at optimizing and auto-tuning the stencil computation on a variety of multi-core architectures, including the GTX 280 GPU. Nukada [12] presented a method of auto-tuning the 3D FFT library on CUDA, while Li [9] looked at optimizing and auto-tuning a CUDA implementation of the GEMM algorithm. Meanwhile, Liu [11] introduced a more general adaptive framework that takes the input parameters and uses the framework to generate the optimal CUDA implementation for the given input.

Our work differs from this body of related work because of our focus on optimizing and developing an auto-tuning framework for the belief propagation algorithm that has not been optimized for CUDA across a large range of possible inputs and GPUs.

11 Conclusions and Future Work

In this paper, we explored methods to optimize a CUDA belief propagation implementation for stereo vision processing. Our results provide insights and results which can be used to optimize a real-life implementation of belief propagation for stereo; such an implementation could be utilized as part of a real-time computer vision system, among other real-world applications.

In the process, we explored the optimization space of using local, shared, and register memory options for data storage on the GPU. It is clear that high-latency local memory accesses should be kept to a minimum, and we explored various options to achieve that goal. We looked at the results of optimizations on the GTX 285, Tesla C870, and GTX 470 GPUs, and discovered that the properties of the target GPU(s) must be taken into account when optimizing a CUDA program. We showed that our optimizations work on two distinct stereo sets with different properties.

In the future, we intend to explore various properties of the CUDA compiler, such as when the compiler will automatically unroll a loop inside a kernel and how the compiler handles register spilling. We plan to run our optimized belief propagation implementations on stereo sets with varying characteristics and with a variety of input parameters, as well as explore how to optimize a belief propagation implementation for 2D motion estimation from a set of sequential images.

References

1. NVIDIA CUDA Programming Guide: Version 3.0. NVIDIA Corporation (February 2010), `http://developer.nvidia.com/cuda`

2. Brunton, A., Shu, C., Roth, G.: Belief propagation on the gpu for stereo vision. pp. 76–76 (June 2006)
3. Callahan, D., Carr, S., Kennedy, K.: Improving register allocation for subscripted variables. SIGPLAN Not. 25(6), 53–65 (1990)
4. Cooper, K.D., Lu, J.: Register promotion in c programs. In: Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI-97. pp. 308–319. ACM Press (1997)
5. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. pp. 1–12. IEEE Press, Piscataway, NJ, USA (2008)
6. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient belief propagation for early vision. Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 1, 261–268 (2004)
7. Grauer-Gray, S., Kambhamettu, C.: Hierarchical belief propagation to reduce search space using cuda for stereo and motion estimation. In: 2009 IEEE Workshop on Applications of Computer Vision (WACV 2009) (Dec 2009)
8. Grauer-Gray, S., Kambhamettu, C., Palaniappan, K.: Gpu implementation of belief propagation using cuda for cloud tracking and reconstruction. pp. 1–4 (Dec 2008)
9. Li, Y., Dongarra, J., Tomov, S.: A note on auto-tuning gemm for gpus. In: ICCS '09: Proceedings of the 9th International Conference on Computational Science. pp. 884–892. Springer-Verlag, Berlin, Heidelberg (2009)
10. Liang, C.K., Cheng, C.C., Lai, Y.C., Chen, L.G., Chen, H.H.: Hardware-efficient belief propagation. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 80–87 (2009)
11. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for gpu program optimizations. Parallel and Distributed Processing Symposium, International 0, 1–10 (2009)
12. Nukada, A., Matsuoka, S.: Auto-tuning 3-d fft library for cuda gpus. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 1–10. ACM, New York, NY, USA (2009)
13. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 73–82. ACM, New York, NY, USA (2008)
14. Scharstein, D., Szeliski, R., Zabih, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. pp. 131–140 (2001)
15. Scharstein, D., Szeliski, R.: High-accuracy stereo depth maps using structured light. Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 1, 195 (2003)
16. Sun, J., Zheng, N.N., Shum, H.Y.: Stereo matching using belief propagation. IEEE Trans. Pattern Anal. Mach. Intell. 25(7), 787–800 (2003)
17. V. Ivanchenko, H.S., Coughlan, J.: Elevation-based stereo implemented in real-time on a gpu. In: 2009 IEEE Workshop on Applications of Computer Vision (WACV 2009) (Dec 2009)
18. Yang, Q., Wang, L., Yang, R., Wang, S., Liao, M., Nistér, D.: Real-time global stereo matching using hierarchical belief propagation. In: British Machine Vision Conf. pp. 989–998 (2006)
19. Yanyan Xu, Hui Chen, R.K.J.L., Vaudrey, T.: Belief propagation implementation using cuda on an nvidia gtx 280. pp. 180–189 (Nov 2009)