A GPU IMPLEMENTATION OF TILED BELIEF PROPAGATION ON MARKOV RANDOM FIELDS

BY

MARIA KOTSIFAKOU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Vikram Adve

ABSTRACT

In this work, we present a parallelized version of tiled belief propagation for stereo matching. The proposed algorithm is implemented in CUDA to leverage parallel processing capabilities of GPUs. In our solution, the original tiled BP algorithm is combined with a number of optimizations specific to parallel programs in CUDA. For the given test inputs, the proposed solution runs in 7.96 milliseconds on Nvidia Tesla C2050, achieving acceptable accuracy with respect to the reference code.

This work has been published in 2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013), winning the MEMOCODE Design Contest 2013 in the adjusted cost-accuracy category. To the best of authors knowledge, this represented the first work in optimizing a parallelized version of the tiled BP algorithm.

After presenting our approach, at selecting an appropriate candidate algorithm for parallelization and implementing in on GPU by applying a series of appropriate optimizations, we discuss the current state of the art on stereo matching, that has been presented since publishing this work.

To my parents, Panos and Nota,
for always loving and supporting me.

To my sister, Annita, my best friend,
for always trying to make me smile.

To my wonderful little brother, Dimitris,
for always showing me that he was proud of me.

ACKNOWLEDGMENTS

I would like to thank my colleagues, Hassan Eslami and Theodoros Kasampalis, for their contribution to this work. Hassan brought the MEMOCODE Contest 2013 to my attention, and both Hassan and Theodoros made working on this project a really fun and rewarding experience.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Stereo matching is an important kernel in computer vision. The goal is to infer depth information for a scene given a stereo image pair. A stereo image pair consists of two images, each depicting a different view of a certain scene as it would be visible from two horizontally displaced locations. Depth information for each pixel can be inferred by comparing the two images of the input stereo image pair and making use of the Parallax effect. This effect also occurs in human vision, and describes the intuitive property that more distant objects appear to be displaced by a smaller amount than more close ones when comparing the two different images of the input stereo pair.

Given a stereo image pair of two images, i.e. two images taken by cameras at different horizontal positions, we aim to infer depth information for each pixel in the left image. Disparity d is a term referring to the difference in the horizontal location of an object in the left and right image of the stereo pair: an object at the position (x, y) in the left image appears at position $(x - d, y)$ in the right image. The disparity of an object can be used to compute its depth using information related to the points where the images were taken and the physical properties of the cameras. Therefore, depth for each pixel can be inferred by comparing the two images of the input stereo image pair and making use of the Parallax effect. The output of the computation is a disparity map, containing a label which represents the estimated depth for each pixel.

The Loopy Belief Propagation algorithm on Markov Random Fields [1] is one of the best-known approaches for stereo matching. This algorithm models stereo matching as an energy minimization problem in the Markov Random Field framework. It assigns a depth value to each pixel by iteratively computing and exchanging messages between neighboring pixels of the image in order to minimize an energy function that depends on the depth assigned to each pixel. Tiled Belief Propagation, an algorithm that performs Belief Propagation with significantly smaller memory and bandwidth requirements, is presented in [2]. [3] presents an alternative message construction mechanism that reduces the complexity of message computation.

The rest of the document is organized as follows: Chapter 2 provides background information about the algorithms we use, and a brief description of the NVidia GPU architecture and programming model that is our target hardware. In chapter 3 we describe the challenges of parallelizing this particular application on top of the target hardware. In chapter 4 we provide a detailed explanation of our implementation of Tiled Belief Propagation on GPUs, that was developed and published as part of the MEMOCODE Design Contest 2013 [4, 5].

Chapter 5 includes an evaluation of our implementation. Chapter 6 discusses the current state of the art on stereo matching that has been presented since the evaluation of this work at the MEMOCODE Design Contest 2013, and chapter 7 concludes.

CHAPTER 2: BACKGROUND

This section describes the work on which we based our parallelized implementation of stereo matching. We first describe the Loopy Belief Propagation method, one of the best approaches to stereo matching, and the Tiled Belief Propagation, that is a variation of Loopy Belief Propagation that is used for our implementation due to its suitability for parallelization. Finally, we give a brief overview of the NVidia GPU architecture and programming model.

2.1 LOOPY BELIEF PROPAGATION

Loopy Belief Propagation [1] models stereo matching as an energy minimization problem in the Markov Random Field (MRF) framework.

Each pixel is represented by a node in a grid graph representing the MRF. The edges of the graph connect nodes that represent neighboring pixels. Let l be the set of all possible labels denoting the inferred depth, and L be the size of l. Each pixel is associated with an L-dimensional data cost, denoting the cost of assigning each label l_p to pixel p, and four L-dimensional incoming messages, each from one of its neighboring pixels. The energy of a particular label assignment is determined by the data cost associated with each node as well as the the sum of smoothness costs of all neighboring node pairs, according to the energy function in equation 2.1.

$$E = \sum_{p} data\_cost_p(l_p) + \sum_{p,q \in neighbors} V_{pq}(l_p, l_q) \tag{2.1}$$

After an initialization phase, the algorithm iteratively computes outgoing messages from every pixel P_{src} to its neighboring pixels P_{dst} by combining the incoming messages from the remaining three neighboring pixels and its local data cost, aiming to minimize the aforementioned energy function.

In the BP-M scheme [6], proposed to accelerate the convergence speed, an iteration is completed in four phases, each one updating messages at one direction (right, left, down, up). An example is shown in figure 2.1, where an iteration of belief propagation on four-pixel image propagates messages first to the right, to the felt, down, and up direction.

Figure 2.2 shows the work performed for one pixel during propagating messages to the right. Incoming messages are read from each of the three neighboring pixels up, down and left. A new message is computed using the local data and the incoming messages, and is

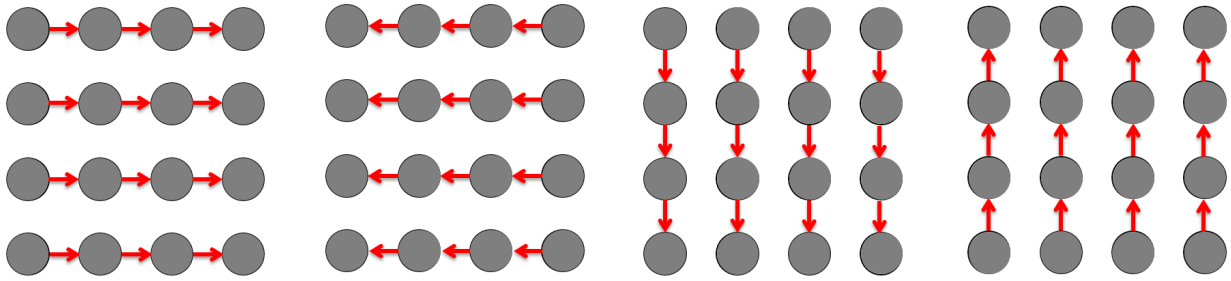


Figure 2.1: Belief Propagation Iteration

sent to the neighboring pixel to the right.

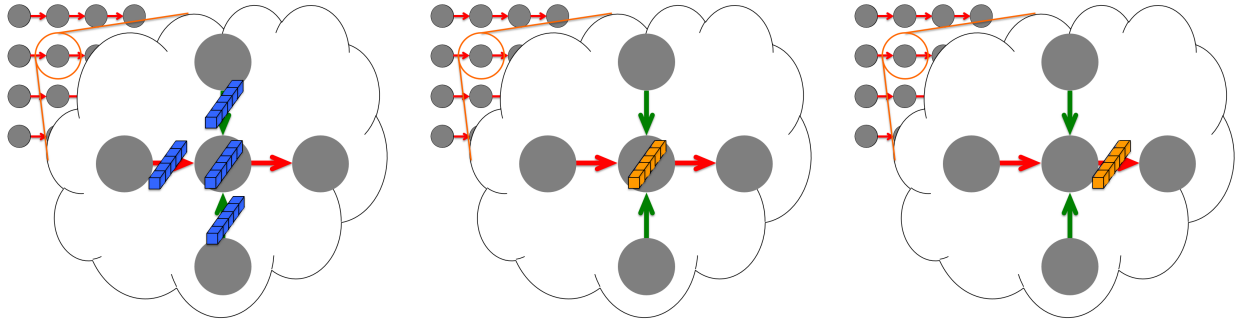


Figure 2.2: Belief Propagation computation by a pixel

2.2 TILED BELIEF PROPAGATION

The original belief propagation implementation computes a vast number of messages among nodes in the network representing the Markov Random Field. For example, according to the reference code of MEMOCODE Design Contest 2013, $L = 16$ - each message is a vector of 16 numbers associated with different labels/color intensities for pixels in the resulting picture of stereo matching process. For each element in the vector, the computation needs all vectors of incoming messages to the node and the data stored in the node itself. Although there is a significant amount of computation involved, accessing the required data from memory is a bottleneck. This memory intensive behavior of the algorithm is mitigated in the tiled belief propagation (BP) method proposed in [2].

In the tiled BP algorithm, the image is divided into square tiles, namely $T_0, T_1, ..., T_n$ (assuming a row-wise ordering of all tiles for demonstration purposes). In each iteration, tiles are processed one by one, first in raster $(T_0, T_1, ..., T_n)$ and then in reverse raster order $(T_n, ..., T_1, T_0)$. This is shown in figure 2.3.

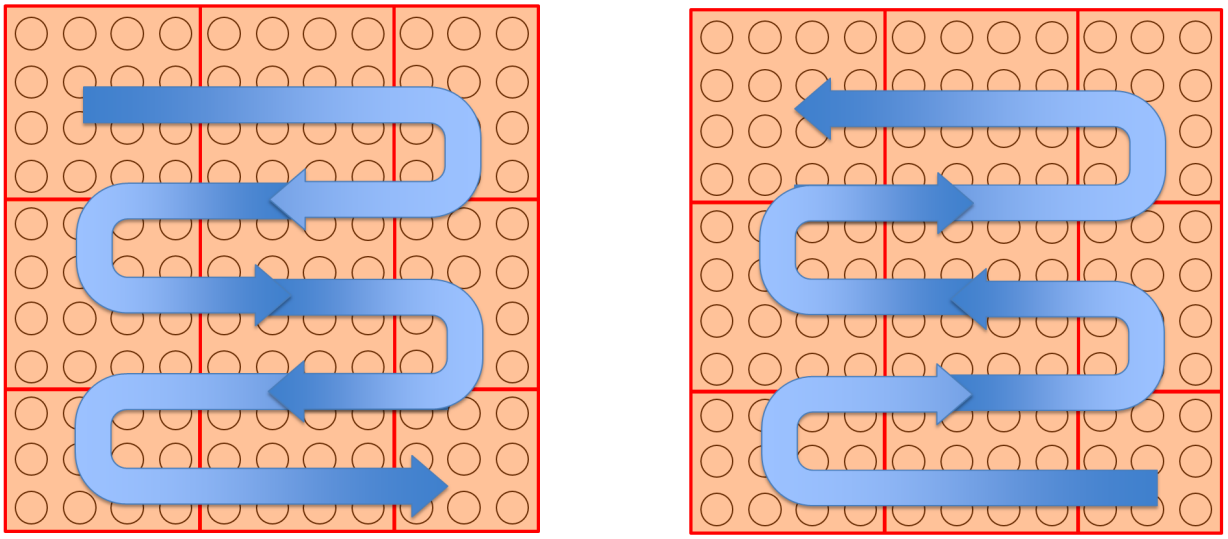


Figure 2.3: Raster scan and reverse raster scan

Beside this outer iteration over all tiles, internal messages for each tile are calculated by applying several iterations (called inner iterations) of BP algorithm. Incoming messages for the pixels that lie in the boundary of a tile, from the direction outwards from the tile, are the boundary messages of the neighbouring image tiles at the corresponding direction. Figure 2.4 demonstrates the work performed by a tile in Tiled BP algorithm. Incoming messages are read from the neighboring tiles, then one or more iterations of BP are performed, and then the outgoing messages are updated.

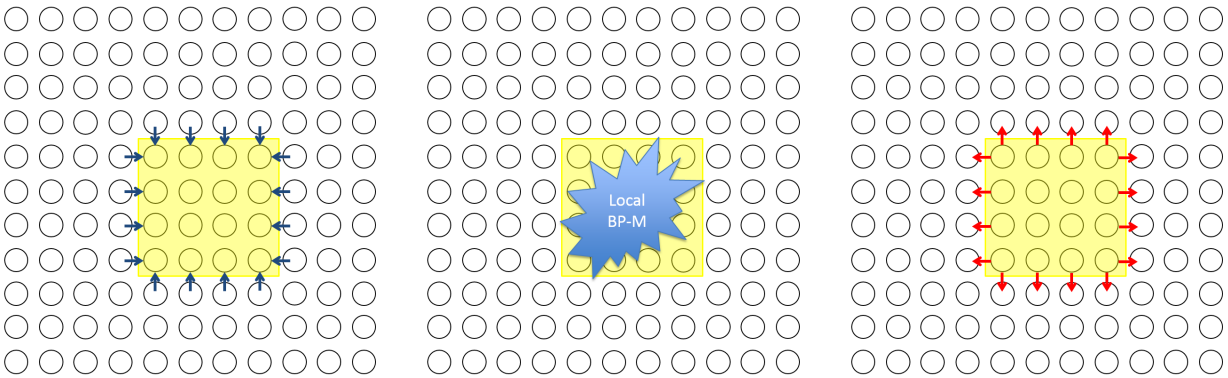


Figure 2.4: One tile in tiled Belief Propagation

2.3 GPUS

Graphics Processing Units (GPUs) are throughput-oriented, highly parallel, many-core architectures that issue a massive number of light-weight threads to exploit data parallelism. NVidia GPUs contain hundreds of cores, called scalar processors (SPs), grouped in streaming multiprocessors (SMs). The threads are organized in thread blocks, and all threads of the same thread block are executed at a single SM. An SM can execute multiple thread blocks at the same time, using dedicated zero-overhead hardware to issue commands from different thread blocks in order to hide the latency of individual operations. While each thread block is completely independent of each other, threads of the same thread block can cooperate. The GPU memory hierarchy includes private (per thread) local memory and registers. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. Figure 2.5 shows the GPU memory hierarchy, simplified to depict the memory components that will be used throughout this document.

The following section provides a description of how these devices can be programmed, using an application programming interface (API) that exposes

2.4 GPU PROGRAMMING

This section gives a brief description of the main concepts related to GPU programming. We give a high level overview of CUDA, and discuss the concepts of kernels, threads, thread blocks, shared memory, and synchronization.

2.4.1 CUDA

CUDA [7] is a parallel computing platform and programming model developed by NVidia, to eanble programming of NVidia based GPUs. CUDA API allows for programming in C / C++.

2.4.2 CUDA Kernel

In the CUDA programming model, a central concept is a kernel. A kernel is a function that will be executed on the GPU, and can be launched by the host code. The kernel function is a single threaded, sequential code, and describes the computation performed by one thread. The (host, C/C++) code invoking the kernel function needs to further specify

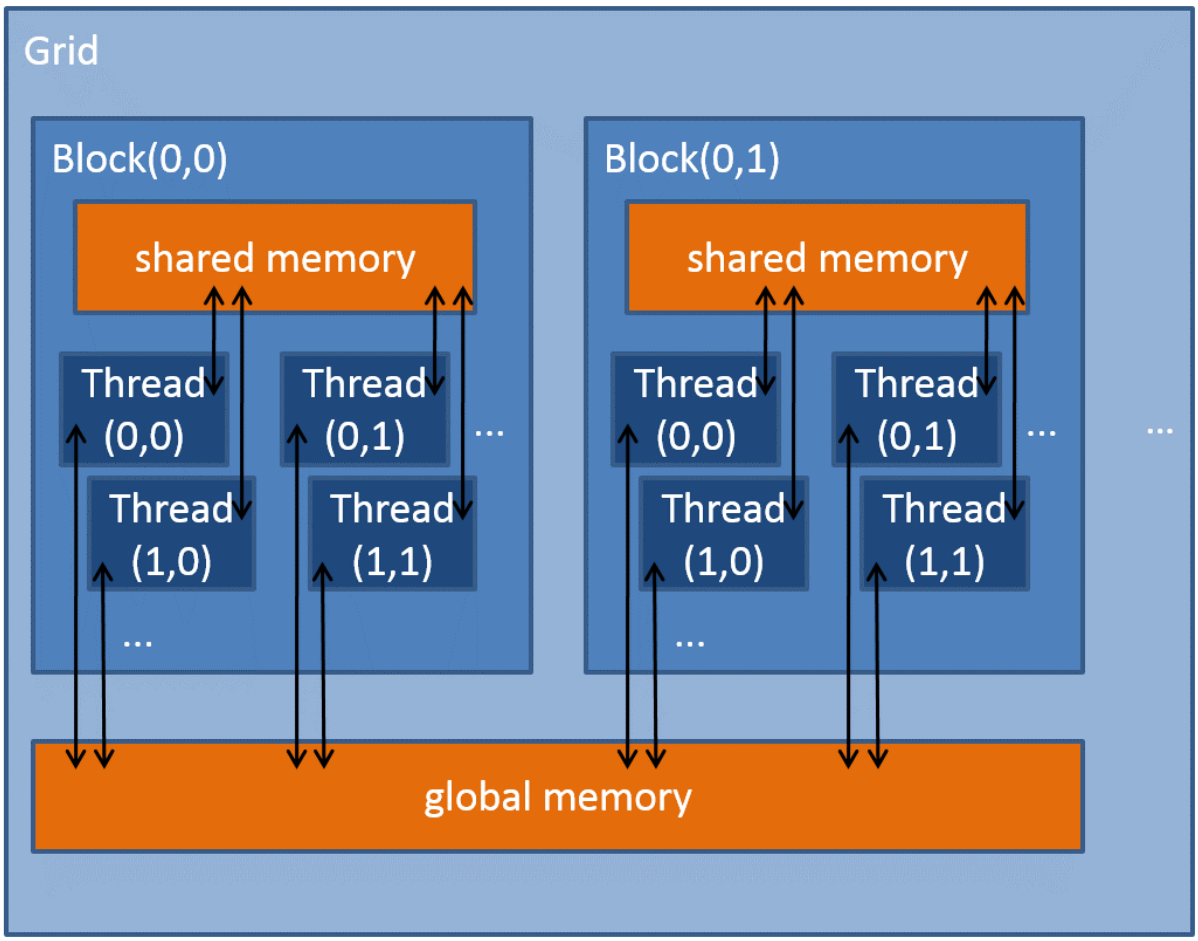


Figure 2.5: GPU memory model

a configuration, that indicates the number of parallel threads to be issued as well as their organization. Listing 2.1 shows a simple GPU kernel function and the code that launches this kernel.

Listing 2.1: CUDA kernel for vector addition and host code to invoke it

```
__global__ void vecAddGPU(float* out, float* in1, float* in2, int N)
{
    // Unique thread index calculation using built-in variables
    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < N) {
      // Only threads within the data range will perform computation
      out[idx] = in1[idx] + in2[idx];
```

Listing 2.1 (Cont.): CUDA kernel for vector addition and host code to invoke it

```
    }
}

void vecAddCPU(float* out, float* in1, float* in2, int N){
  // Thread Block size
  int numThreadsPerTB = 256;
  // Total number of threads and thread blocks is data dependent
  int numTBs = (N-1)/numThreadsPerTB + 1;

  // Call kernel. This is an asynchronous operation.
  vecAddGPU<<< numTBs, numThreadsPerTB >>>(out, in1, in2, N);

  // Wait for execution of kernel to be completed.
  cudaDeviceSynchronize();
}
```

2.4.3 Thread Hierarchy

As mentioned in 2.4.2, a kernel only describes the computation performed by a single thread, and the code that calls the kernel needs to provide additional information about its configuration.

When a CUDA kernel is called, it is launched in thread blocks, each consisting of a number of threads, as shown in figure 2.6.

This configuration is unknown to the kernel function, and it is the calling code's responsibility to specify it. The kernel function however can query the configuration parameters.

In the given example, the kernel function is intended to perform an addition of the input arrays in1 and in2 of size N and write the output on array out. However, the kernel itself does not know how many threads would be launched. The total number of threads required to complete the computation is data dependent, and it is provided at kernel launch time by the host code.

The computation contained in the kernel function body describes the work performed by a single thread, and performs the addition of the two input elements in the corresponding location of the input arrays and stores the result to the appropriate location in the output array. Each thread that executes a kernel is given a thread ID, that is unique to the thread

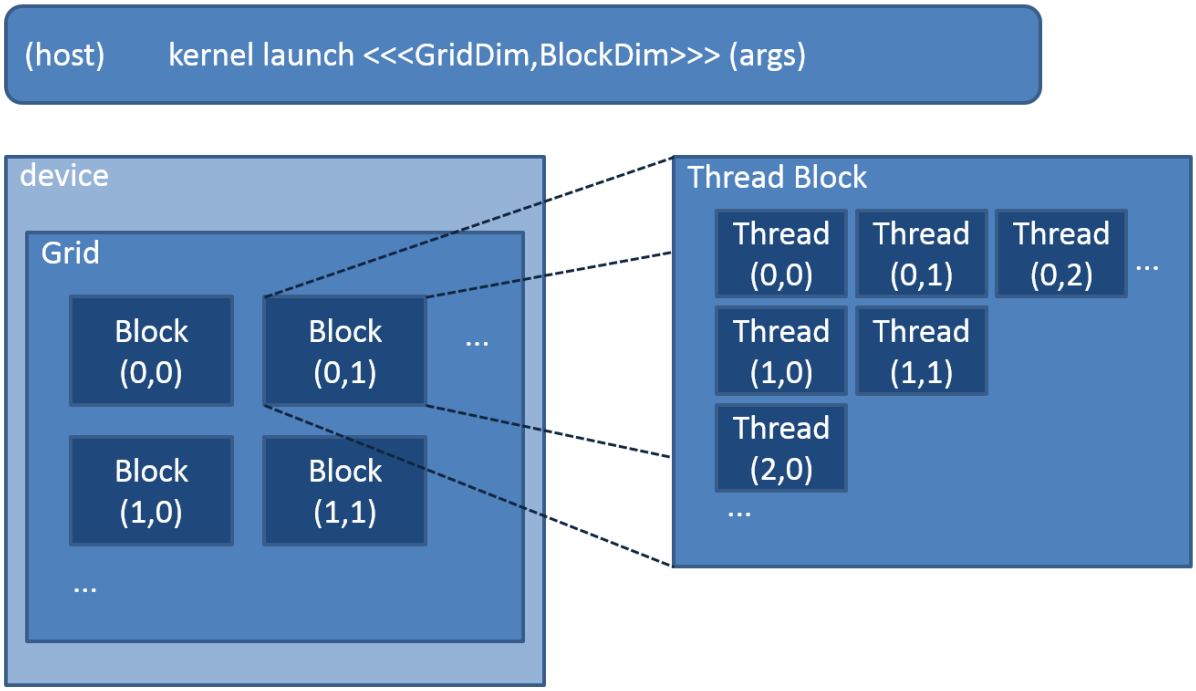


Figure 2.6: GPU memory model

within the thread block it belongs to. Similarly, each issued thread block is given a unique block ID, also unique. These, as well other configuration parameters such as thread block size, are accessible within the kernel through builtin variables. In this example, threadIdx , blockIdx, blockDim are used to access the unique thread index, block index, and the block dimensions, i.e. the number of threads per thread block, to create a unique global identifier foe every thread.

For convenience, the allowed configurations need not be one dimensional. Thread blocks can be configured as one-dimensional, two-dimensional, or three-dimensional, and similarly thread blocks can also be organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. This provides a natural way to decompose computation across the elements in a domain such as a vector, matrix, or volume. The builtin variables threadIdx, blockIdx, etc are 3-component vectors, so that threads and thread blocks can be identified in each dimension of the thread block and grid respectively.

2.4.4 Memory Hierarchy

The memory model of GPUs includes

- Global memory
  An off-chip memory accessible by all threads

- Shared memory
  A fast on-chip scratchpad memory that is shared between threads of the same thread block. It is allocated at a per thread block basis, and all threads of a thread block have access to the same shared memory. Threads within the same thread block can cooperate by sharing data through shared memory and synchronizing their execution to coordinate memory accesses. This is done using a barrier instruction, forcing threads to wait until all threads of the thread block reach the barrier.

  A common optimization that uses this resource is cooperative loading of data in shared memory by threads of a thread block, and using the loaded data in shared memory instead of accessing the large but slow global memory. This allows threads to access data that cooperating threads loaded instead of issuing more memory requests, effectively reducing the required memory bandwidth and increasing the amount of computation that can be performed per loaded data.

CHAPTER 3: CHALLENGES

To parallelize an application for a GPU, there are two important considerations. Firstly, there are multiple algorithms that aim to solve one problem, each with different advantages and disadvantages. Secondly, the target hardware has differing features, consequently providing different advantages and at the same time imposing limitations. One has to consider these two intertwined in order to come up with a good implementation of stereo matching for GPUs.

3.1 CONSIDERATIONS FOR GPU PERFORMANCE

GPUs are a highly parallel, massively threaded, throughput oriented accelerator. It can provide high speedups due to massive number of lightweight parallel threads, high memory bandwidth, and hiding of latency by context switching. Due to these features, there are factors that affect the performance that can be achieved, different compared to CPU programming. The following is a list of the factors that we have found to be relevant in our effort to parallelize this application.

- Effective utilization of memory bandwidth
  GPUs provide very high memory bandwidth, in order to support the high number of parallel threads that compete for access to the data they require for their computation. Being able to have a high number of threads executing on the GPU is essential to achieving good performance. Albeit, it means that the memory bandwidth needs to be carefully utilized. The computation to memory operation ratio is a metric that expresses who many operations can be executed for any given memory operation, and can be used in order to determine how effectively the memory bandwidth is utilized. A high computation to memory operation ratio means that once a memory operation is complete, multiple operations can be performed and thus the compute units will be utilized. A low computation to memory operation ratio indicates that only a few operations may be completed per memory operation, meaning that the compute units will be idle while waiting for the memory requests to be served.

- Synchronization
  Synchronization is in general a costly operation, and should always be used with care when programming a parallel system. Specifically on a GPU, some sorts of synchronization can be very expensive, while others are relatively cheap. One type that is

relatively cheap is synchronization at the thread block level. Threads that belong to a single thread block can synchronize at any point during their execution using the instruction *__syncthreads*(). This, combined with usage of the fast shared memory, can be used to allow groups of threads to cooperate. Global synchronization, i.e. synchronization of all the threads that have been issued to execute a CUDA kernel, is difficult and expensive. In the general case, threads may only synchronize at the end of the kernel execution, thus a new kernel launch is required to enforce a global synchronization. Under certain assumptions and limitations, it is possible to implement a global synchronization scheme within a kernel [8]. In any case, algorithms that require global synchronization would need to be refactored, usually in a significant way.

3.2 BELIEF PROPAGATION ALGORITHM

As described in section 2, stereo matching can be represented as a Markov Random Field, with nodes being the image pixels and edges describing a 4 connected grid between neighboring pixels, and solved by applying belief propagation on Markov Random Fields. However, this algorithm demonstrates some features that have been identified as limiting factors in GPU performance or inhibit parallelization in general.

- Serialization
  In Loopy Belief Propagation using the BP-M scheme, the propagation of messages to each direction - right, felt, up, and down - is completely sequential. This significantly limits the available parallelism.

- Many global synchronization points
  Global synchronization is required every time a sweep to the right, left, up, or down is completed, in order to respect the order of computation described in Loopy Belief Propagation using the BP-M scheme. It would be difficult to achieve good performance with many global synchronization points when combined with the imposed serialization, described above.

- High memory bandwidth requirements
  In belief propagation, each node performs a computation involving local data and messages acquired by three neighbor nodes in order to compute the message to be send to the remaining neighbor node. Because a node computes a message only once per sweep, data reuse will not occur. Thus, the total amount of data transfer required is very high, making this a memory bound algorithm.

CHAPTER 4: PARALLELIZATION STRATEGY

4.1 ALGORITHM SELECTION

As detailed in section 3.2, the Loopy Belief Propagation algorithm has characteristics that make it difficult to parallelize for GPUs. We consider instead the tiled BP algorithm.

The tiled BP algorithm is a good fit for the GPU programming model because it has three characteristics:

- Data parallel The tiled BP algorithm is highly data parallel. Each tile performs an identical computation on a different data set. Therefore, we can exploit the vast computation power of GPUs which have the potential to perform well on data parallel programs.

- Reduced Memory Bandwidth requirements The tiled BP algorithm has significantly smaller memory bandwidth requirements compared to other BP algorithms. All the computations for all the inner iterations of one tile for a particular outer iteration requires acquiring information about the boundary messages just once, in the beginning of the tile processing. Therefore, the required memory bandwidth is significantly reduced, which is a major factor in achieving good performance in GPUs.

- Regular access pattern The tiled BP algorithm demonstrates a regular memory access pattern. This feature makes tiled BP an outstanding choice for GPU implementation as by choosing appropriate tile sizes we can exploit this behaviour in order to make the best use of shared memory. We can simply load all required data in the shared memory of GPU first, and then apply BP-M method in the inner iterations. Hence, the computation for each tile is done in two phases: loading data from the global memory to the shared memory, then running several iterations of BP on the data loaded into the shared memory.

Note that it is not the presence of one of these features that make this algorithm a good target for parallelization on GPUs, but the combination of all three. For example, the original BP algorithm also is data parallel and presents a regular access pattern. However, it does have a significantly higher memory bandwidth requirement, making the regular access pattern more difficult to difficult to utilize efficiently.

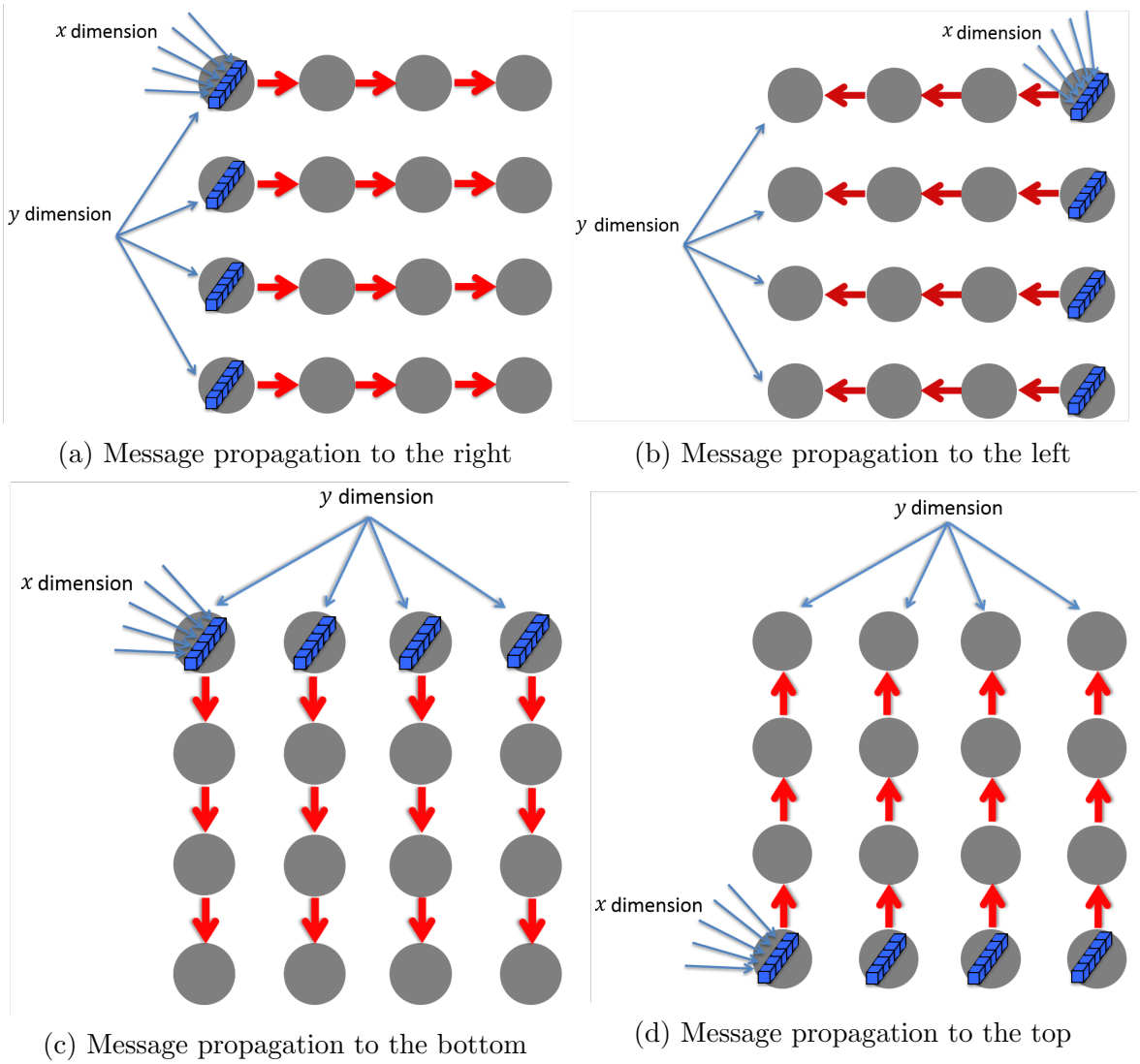
4.2 PARALLELISM DECOMPOSITION IN THREADS AND THREAD BLOCKS

Our proposed solution exploits parallelism in two levels: parallelizing computations for different tiles (coarse granularity), and parallelizing computation inside a tile (fine granularity).

We assign a thread block to an image tile of square shape. For the fine-grain level parallelism, we attempt to maximize the parallelism available within an image tile. For the coarse-grain level parallelism, we attempt to execute the computation on as many image tiles (e.g. by as many thread blocks) as possible in parallel.

As mentioned, each thread block is assigned to an image tile. The thread block has a number of threads organized in a 2-dimensional grid. In the x dimension of the grid, threads are responsible for computing elements of message vectors. Since each message vector has 16 elements (as in the reference code), the size of the x dimension is 16. In the y dimension of the grid, threads propagate messages in the different directions. Threads in the y dimension behave differently, depending on the direction of the message propagation, depicted in figure 4.1. In the right (similarly, left) message propagation, since computation for different rows is independent, different threads compute messages for different rows, starting from the leftmost (similarly, rightmost) element and moving towards the right (similarly, left). Likewise, computation for different columns is independent in up/down message propagation phase. Having said that, the size of y dimension is the same as the tile size.

As in the original tiled BP method, there must be an ordering for computation on different tiles (raster and reverse raster order). This ordering imposes a sequential dependency in the computation performed on different image tiles. However, based on the tiled BP algorithm, in order to compute messages inside a tile, we only need boundary incoming messages for that tile. Thus, computation in the wavefront model (diagonal by diagonal, as depicted in figure 4.2) gives us an opportunity for parallelization. In other words, when computing on different tiles in wavefront model, computations for the tiles on the diagonal are independent and can be done in parallel. The only subtlety in this model is that computation on different diagonals should be ordered, from the first to the last when modeling the raster order and in reverse when modeling the reverse raster order. Therefore, we need a global synchronization point after completing the computation of all tiles in a diagonal. We describe the global synchronization scheme in section 4.4.



(a) Message propagation to the right

(b) Message propagation to the left

(c) Message propagation to the bottom

(d) Message propagation to the top

Figure 4.1: A thread block performing one inner iteration of Belief Propagation

4.3 SHARED MEMORY

Within each image tile, there is a large amount of data that needs to be accessed more than one times. In particular,

- messages computed by groups of threads for an image pixel need to be accessed by groups of threads that are performing computation on the neighboring pixels.
- local data cost labels of each pixel need to be accessed multiple times, one per direction and proportionally to the number of inner iterations.

To reduce the required memory bandwidth and make use of the data reuse, we utilize the shared memory to load all required data for each tile at the beginning of the tile's processing.

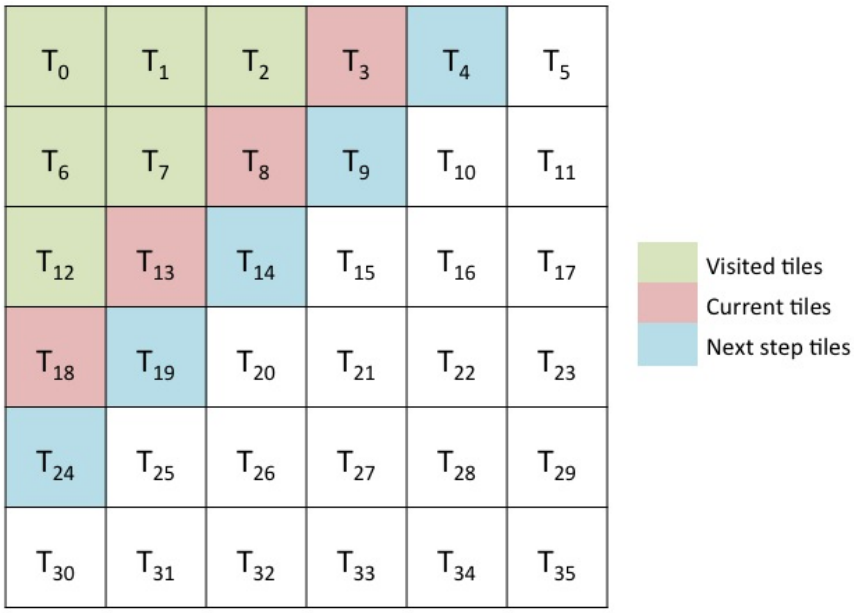


Figure 4.2: Wavefront computation model.

The computation requires all messages inside and on the boundary of the tile for each of the four directions, and all data costs for pixels within the tile. A byte is used to represent each pixel's data cost element, and an unsigned integer for each element in a message. Therefore, the shared memory requirements of each thread block processing a square image tile of size TS is

$$4 \times TS^2 \times 16 \times (sizeof(unsigned\,int)) + TS^2 \times 16 \times (sizeof(byte)) \tag{4.1}$$

All threads collaboratively load the data from global to shared memory. We have implemented the data transfers to shared memory so that consecutive threads in a thread block (in the grid of threads, threads with consecutive IDs in the x dimension) have been assigned to transfer consecutive data from global memory. Such data transfer patterns can be optimized on GPUs, by coalescing the memory requests. Memory coalescing is key in efficiently utilizing the GPU memory bandwidth.

Similarly, some data need to be written back, from shared to global memory, namely the boundary messages that have been computed for the neighboring image tiles. This data transfer is much smaller in size, nevertheless also makes use of memory coalescing.

4.4 GLOBAL SYNCHRONIZATION

Section 4.2 describes the requirement for a global synchronization point between processing different diagonals of the input.

We can implicitly enforce this synchronization by having each diagonal of the input that we encounter in a wavefront being processed by a separate kernel launch from host CPU. However, this method has a huge impact on the overall performance, as a kernel launch is a costly operation.

Instead, we use the lock-free barrier synchronization implementation for GPU proposed in [8]. This allows for all the computation to be performed in a single kernel launch with a barrier at the end of each diagonal processing. This approach can significantly improve performance by reducing the required kernel launches.

The following listing 4.1 from [8] demonstrates the barrier implementation.

Listing 4.1: GPU barrier synchronization

```
// GPU lock-free synchronization function
__device__ void __gpu_sync(int goalVal, int *Arrayin, int *Arrayout)
{
  //thread ID on a block
  int tid_in_block = threadIdx.x * blockDim.y + threadIdx.y;
  int nBlockNum = gridDim.x*gridDim.y;
  int bid = blockIdx.x * gridDim.y + blockIdx.y;

  // only thread 0 is used for synchronization
  if (tid_in_block == 0) {
    Arrayin[bid] = goalVal;
  }

  if (bid == 1) {
    if (tid_in_block < nBlockNum) {
      while (Arrayin[tid_in_block] != goalVal) {
         // busy wait ......
      }
    }
    __syncthreads();
```

Listing 4.1 (Cont.): GPU barrier synchronization

```
21
22    if (tid_in_block < nBlockNum) {
23      Arrayout[tid_in_block] = goalVal;
24    }
25  }
26
27  if (tid_in_block == 0) {
28    while (Arrayout[bid] != goalVal) {
29      // busy wait ......
30    }
31  }
32  __syncthreads();
33 }
```

The barrier is invoked as follows $\_\_gpu\_sync(goalVal{+}{+}, Arrayin, Arrayout)$, with *Arrayin* and *Arrayout* having one element per thread block, and *goalval* being given a different value per invocation. At a high level, the global synchronization is enforced by:

1. One thread per thread block updating its thread block's *Arrayin* entry with the new *goalval*.

2. Having one thread block whose thread are mapped to existing thread blocks using their computed thread indices, and having them wait ($\_\_syncthreads()$ in line 20) until *Arrayin* entries have been updated to the *goalval*. That can only occur when all thread blocks have invoked the global barrier, thus one of their threads updated the thread block's *Arrayin* entry. At that point, the *Arrayout* entries for all thread blocks are updated to the new *goalval*.

3. Having the threads of each thread block wait ($\_\_syncthreads()$ in line 33) until the entry in *Arrayout* corresponding to that thread block is equal to *goalval* and the first thread is allowed to reach this instruction.

With the GPU barrier synchronization, there is a possibility to result in a deadlock due to lack of resources. This is due to the fact that the resources required by a thread block are acquired once a thread block has started executing, and are not released until it completes its execution and retires. That means that even if its execution needs to stall due to a global barrier, the resources will not be released, thus not allowing them to be utilized by any

other thread block. That can create a deadlock, where all thread blocks are waiting on the global barrier and no resources are available to enable other thread blocks to execute and reach the global barrier, preventing all others to continue their execution. A safe approach to avoid deadlock is to limit the number of thread blocks in a kernel launch to the number of Streaming Multiprocessors (SM) in the device. Restricting the number of thread blocks to the number of SMs ensures that all thread blocks will be able to acquire the resources that they need to execute and reach the barrier.

Given the limitations on shared memory size of the available devices at the time of the MEMOCODE Design Contest 2013, this did not impose any restriction on our design. We load a considerable amount of data to shared memory for each tile, hence allowing just one thread block to execute on an SM. Using the equation 4.1, and given the fact that the maximum amount of shared memory available on these devices is 48K, the tile size can be at most 13. Therefore, the size of the y dimension of the thread block grid is 13. As mentioned before the size of the x dimension is 16, which results in $13 \times 16 = 192$ threads in a thread block. This is a small to reasonable number of parallel threads to be issued per thread block, therefore we do not explore options that would allow more than one thread block to be executed by one SM, and thus disallow the use of the GPU barrier synchronization.

Therefore, we launch a kernel with as many thread blocks as there are SMs on the target device. For diagonals with tiles fewer than or equal to the number of thread blocks, the thread blocks simply process the existing tiles. For diagonals which contain more tiles than the number of thread blocks, a static scheduling policy is implemented manually. It is based on the principle of having a number of "virtual" thread blocks $VBlocks$, equal to the number of image tiles on the diagonal, executed by a fixed number of "physical" thread blocks $PBlocks$ (the ones that have been launched). Virtual thread blocks are assigned to physical thread blocks in a round robin scheme, namely a physical thread block with index $pIdx$ will be responsible for the computation of virtual thread blocks $pIdx, pIdx + PBlocks, pIdx + 2 \times PBlocks, ...$, Its computation is completed when the computed virtual thread index exceeds the number of virtual thread blocks.

4.5 OTHER OPTIMIZATIONS

By applying all the aforementioned optimization techniques, the tiled BP method becomes a compute intensive algorithm.

In order to further reduce the running time, we applied two additional optimizations.

- Fast message computation:

We optimized the way messages are computed. We get a slight performance gain by using the fast message computation proposed in [3]. We use a small additional amount of shared memory to store the intermediate results shared by the threads during this computation, remaining below the limit of 48K for an image tile size of 13.

- Removing unnecessary synchronization points:
  We noticed that in order to follow the exact tiled BP algorithm mentioned in [2] and message computation in [3], we would have to synchronize all threads within one tile at each step of the message exchange and in points within the message computation. Through manual analysis of the code and relying on the features of GPU architectures available during our development, we found that only two points of synchronization were required.

  This is due to the fact that on the Nvidia GPU architecture, groups of threads are scheduled to execute in the granularity of warps (32 threads). On the GPUs available during our development (up to, not including, devices of compute capability[1] 7.x and higher), threads of a warp are executed in lock-step. Threads are assigned to warps in a predictable way, by x being the least significant dimension and y after it. This allowed us to ensure that threads processing data of one pixel (the first 16 or the last 16 threads with the same identifier in the y dimension) are executed in lock-step, i.e. the threads operating on one pixel were always synchronized due to belonging to the same warp - that is referred to as warp-synchronous code. With manual analysis of the code, we determined that the only operation for which thread block-wide synchronization was needed was computing the sum of a per pixel data cost and incoming messages for a label. A $\_\_syncthreads()$ was needed before and after this computation, to ensure that (1) all threads use the updated value computed for the current pixel and (2) no thread updates the computed value before it is used by other threads that might access it.

  Additionally, in our experiments during the evaluation, we found that even these two synchronization points were not indeed necessary, only causing negligible variation in the resulting accuracy, and were therefore also removed.

[1] a device's set of computation-related features

4.6 COMPLETE ALGORITHM

The parallelized algorithm of tiled BP, as described in the previous sections, is shown in pseudocode in algorithms 4.1, 4.2, 4.3, 4.4. We aim to provide a high level overview of our approach, thus omitting details such as additional function parameters, exact index calculations and computations, boundary condition checking etc. The tiled belief propagation kernel is described in 4.1. It iterates over the input image data $OUTER\_ITERATIONS$ times, each time invoking a raster and a reverse raster diagonal pass (4.2 and 4.3 respectively). Both passes invoke 4.4 to perform $INNER\_ITERATIONS$ of belief propagation within each tile of the diagonal, imposing the necessary synchronization between them. The exact message computation is omitted, as we use the message computation scheme in [3]. Finally, algorithm 4.5 shows the kernel invocation, demonstrating it being configured with the number of thread blocks equal to the number of SMs on the target device.

```
   Data: imageData,                                  /* Data costs for all pixels */
   rightB, leftB, upB, downB,                        /* boundary messages */
   Arrayin, Arrayout,          /* used for the barrier synchronization */
   Result: bestAssignment                            /* Label assignment */
1  Function TiledBP:
2      __shared__ TILE tile ;          /* shared memory allocated for a tile */
3      int goalVal ← 1 ;  /* Goal value for the GPU barrier synchronization */
4      for to ← 1 to OUTER_ITERATIONS do
5          rasterDiagonalPass(tile, imageData, rightB, leftB, downB, upB, Arrayin,
            Arrayout, INNER_ITERATIONS);
6          reverseRasterDiagonalPass(tile, imageData, rightB, leftB, downB, upB,
            Arrayin, Arrayout, INNER_ITERATIONS, to, bestAssignment);
7      end
```

Algorithm 4.1: Tiled BP kernel.

```
Data: tile,                          /* Image tile data on shared memory */
imageData,                                  /* Data costs for all pixels */
rightB, leftB, upB, downB,                          /* boundary messages */
Arrayin, Arrayout,              /* used for the barrier synchronization */
INNER_ITERATIONS,       /* number of inner BP iterations to perform */
1  // GPU function, performing the iterations of BP on an image tile
2  Function rasterDiagonalPass:
3      for diag ∈ Diagonals in increasing order do
4          for all tiles assigned to this thread block do
5              /* Load tile data costs and boundary messages from global to
                 shared memory                                            */
6              tile.readMemory(imageData, rightB, leftB, downB, upB);
7              __syncthreads();
8              tile.innerBP(INNER_ITERATIONS);
9              // Write boundary messages to global memory
10             tile.writeMemory(rightB, leftB, downB, upB);
11             __syncthreads();
12         end
13         __gpu_sync(goalVal++, Arrayin, Arrayout);
14     end
```

Algorithm 4.2: GPU function performing raster diagonal pass.

Data: tile, /* Image tile data on shared memory */
imageData, /* Data costs for all pixels */
rightB, leftB, upB, downB, /* boundary messages */
Arrayin, Arrayout, /* used for the barrier synchronization */
INNER_ITERATIONS /* number of inner BP iterations to perform */
currentOutetIteration /* index of current outer BP iteration */
Result: bestAssignment /* Label assignment */

```
1  // GPU function, performing the iterations of BP on an image tile
2  Function reverseRasterDiagonalPass:
3      for diag ∈ Diagonals in decreasing order do
4          for all tiles assigned to this thread block do
5              /* Load tile data costs and boundary messages from global to
                  shared memory                                              */
6              tile.readMemory(imageData, rightB, leftB, downB, upB);
7              __syncthreads();
8              tile.innerBP(INNER_ITERATIONS);
9              if currentOutetIteration ≠ OUTER_ITERATIONS then
10                 /* Write boundary messages to global memory in all but the
                      last iteration                                         */
11                 tile.writeMemory(rightB, leftB, downB, upB);
12             else
13                 /* In the last iteration, compute best label assignment by
                      minimizing the cost.  This is completed in shared memory
                      and written back to global memory                      */
14                 tile.finalize(bestAssignment);
15             end
16             __syncthreads();
17         end
18         __gpu_sync(goalVal++, Arrayin, Arrayout);
19     end
```

Algorithm 4.3: GPU function performing reverse raster diagonal pass.

```
Data: tile,                                  /* Image tile data on shared memory */
INNER_ITERATIONS          /* number of inner BP iterations to perform */
1 // GPU function, performing the inner iterations of BP on an image tile
2 Function innerBP:
3     for ti ← 1 to INNER_ITERATIONS do
4         tile.compute(RIGHT, ...);
5         tile.compute(LEFT , ...);
6         tile.compute(DOWN , ...);
7         tile.compute(UP , ...);
8     end
```

Algorithm 4.4: Inner BP GPU function.

```
Data: imageData,                                /* Data costs for all pixels */
rightB, leftB, upB, downB,                              /* boundary messages */
Arrayin, Arrayout,                 /* used for the barrier synchronization */
Result: bestAssignment                                   /* Label assignment */
1 // Host CPU function, performing the tiled BP kernel invocation
2 Function main():
3     // setup
4     ...
5     // Tiled BP kernel invocation
6     dim3 block_dim(16, 13);          // number of labels in x dimension, 13 in y
          dimension
7     tiledBP <<< NUMBER_OF_SMs, block_dim >>> (imageData, rightBorder,
       leftBorder, upBorder, downBorder, Arrayin, Arrayout, bestAssignment)
8     // read result from device
9     ...
```

Algorithm 4.5: Tiled BP kernel invocation.

CHAPTER 5: EVALUATION

5.1 EXPERIMENTAL SETUP AND RESULTS

We present the evaluation of our solution to stereo matching that was performed for the MEMOCODE Design Contest 2013. We participated in the cost-adjusted-performance category, aiming to achieve the best performance adjusted to the cost of the underlying hardware platform, with the requirement of achieving at least 20% accuracy compared to the ground truth on the provided benchmarks. The judging decision for the winning solution on this category took into consideration the inaccuracy relative to the reference BP algorithm, adjusting the achieved performance according to the formula

$$adjustedRuntime = measuredRuntime \times \frac{numberOfMismathedLabels}{11870} \tag{5.1}$$

where 11870 is the number of mismatched labels inferred by the reference BP implementation.

For the evaluation of our implementation, we tuned the number of outer and inner iterations aiming to maximize the formula 5.1. We achieved that by setting both outer and inner iterations for our algorithm to 1. This means that just one raster, and then one reverse raster order is used in computation on tiles, and that for each tile only one iteration of BP-M is used.

With this configuration we achieve better accuracy (15716 mismatches compared to ground truth) compared to the given reference code (17743 mismatches) for the tsukuba test case, and acceptable accuracy (16713 mismatches, less than 20% worse compared to the reference code as per the contest requirements) for the test case used for judging, red barrel. We tested our implementation on two devices. The devices and their price (as used for judging in the design contest), as well as the number of SMs per device, are shown in table 5.1. The execution time and accuracy achieved on each device for both the tsukuba and the red barrel test cases are shown in tables 5.2 and 5.3 respectively.

Table 5.1: System Configurations used for Stereo Matching testing, 2013 pricing

Device	**Number of SMs**	**Cost (USD)**
Tesla C2050	14	1350
CTX 680	8	500

Table 5.2: Execution Time and Accuracy for Stereo Matching on TSUKUBA test case

Device	**Execution Time (ms)**	**Number of Mismatches**
Tesla C2050	7.96	15716
CTX 680	9.26	15716

Table 5.3: Execution Time and Accuracy for Stereo Matching on RED BARREL test case

Device	**Execution Time (ms)**	**Number of Mismatches**
Tesla C2050	7.95	16713
CTX 680	9.24	16736

5.2 DISCUSSION ON ADJUSTMENTS FOR NEWER HARDWARE FEATURES

Our algorithm has been to achieve good performance and accuracy using the hardware released by 2013. New features in the CUDA programming model as well as changes to the target devices would need to be considered for porting our algorithm to newer GPUs. Specifically, a key design decision needs to be reevaluated, the choice to use a single kernel and a GPU global synchronization scheme instead of separate CUDA kernels for diagonal processing.

- Newer GPUs provide a larger amount of shared memory, up to 92K. However, only 48K may be allocated per thread block, not allowing us to increase the image tile size and the associated number of threads per thread block. However, that means that there would be enough shared memory resources to allow two thread blocks to execute on one SM while executing the tiledBP kernel. Since the limiting resource in our kernel is shared memory, our approach of issuing one thread block per SM to ensure the absence of deadlock would lead to massive underutilization of the GPU. The penalty of this needs to be compared to the penalty of using the end of a kernel as a synchronization point and using multiple kernel launches, one per diagonal, to perform the raster and reverse raster diagonal scans.

- Older versions of CUDA (up to CUDA 4) allow a kernel to be launched only from host code. Versions 5.0 and later lift this restriction, allowing kernels to be launched from an executing GPU kernel as well (CUDA dynamic parallelism [7]). This capability is supported on devices of compute capability 3.5 and above. The kernel launch is still an expensive operation, as it at least includes issuing the grid of the thread blocks to be issued and managing their execution. However, this is an additional alternative to

the gpu global barrier and scheduling scheme we implement.

- Independent Thread Scheduling [7], introduced for devices with compute capability 7, allows full concurrency between threads, regardless of warp. Applications that rely on warp-synchronicity need to be revisited, and to ensure correct execution of warp-synchronous code we need to insert the warp-wide barrier synchronization, $\_\_syncwarp()$ in points where we relied on warp-synchronicity. While this may not cause a significant overhead, due to it imposing synchronization points on code that we had already expected to be executed synchronously, we still insert additional, albeit lightweight, synchronization on a target device that supports full concurrency.

CHAPTER 6: STATE OF THE ART

Since the publication of this work at the MEMOCODE 2013 design contest, other approaches have been presented for stereo matching. This section presents the current state of the art on stereo matching.

A ranking of stereo matching algorithms is available at the Middlebury Stereo Vision Page [9]. At the time of writing, the top performing published stereo matching algorithm is by Taniai *et al.* [10]. This algorithm performs 3D Label Stereo matching, i.e. uses three dimensional labels instead of one dimensional (scalar) labels [11]. It is an approach based on graph cuts. In such methods [12, 13, 14, 15], a graph uses pixels as nodes and the capacity of edges is defined as a function of the smoothness cost of adjacent nodes. The disparity map is determined as the minimum cut of the maximum flow in the graph. In particular, they extend the traditional local expansion algorithm described in [13] for optimizing MRF models with a continuous label space using randomized search.

PatchMatch [16, 17] is an inference algorithm using spatial label propagation. In PatchMatch, each pixel is updated in raster-scan order and its label is propagated to next pixels as their candidate labels. In particular, Slanted Patch Matching [18] is a local matching, built on the concept of a support window - a window centered on a pixel of the reference image - being displaced on the second image until the matching point, which minimizes the color dissimilarity. However, there is an assumption in this procedure, all pixels within the support window have constant disparity, which does not hold for slanted surfaces; they can be thought of as approximated by many fronto-parallel planes. Slanted Patch Matching uses this insight and at each pixel onto which the support window is projected upon estimates an individual 3D plane. Insights from [16, 17, 18] have been utilized as well in [10].

Neural Networks have also been used in stereo matching. Zbontar *at al.* [19] have presented a method for extracting depth information from a stereo image pair, that as its first step uses a convolutional neural network to compute the matching cost. The CNN learns a similarity measure on small image patches. The CNN was trained in a supervised manned in a binary classification data set of similar or dissimilar pairs of image patches. Its output is used to compute the stereo matching cost, with post processing steps to follow as in other stereo matching approaches.

CHAPTER 7: CONCLUSION

In this work we presented a GPU implementation of an algorithm for stereo matching. Our solution is based on the belief propagation algorithm on Markov Random Fields and more specifically on the tiled belief propagation algorithm, a tile-based variant proposed in the literature. Our GPU implementation exploits the reduced memory and bandwidth requirements of tiled belief propagation and introduces two levels of parallelism: both coarse-grained parallelism for computations of different tiles and fine-grained parallelism for computations inside a tile. Also, various GPU-specific optimizations are applied, such as usage of the shared memory to benefit from data reuse, as well as minimal usage of synchronization primitives to increase the level of parallelism. The evaluation demonstrates that our implementation was able to achieve acceptable accuracy in less than 10 milliseconds.

REFERENCES

[1] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother, "A Comparative Study of Energy Minimization Methods for Markov Random Fields with Smoothness-Based Priors," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 6, pp. 1068–1080, June 2008. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2007.70844

[2] C.-K. Liang, C.-C. Cheng, Y.-C. Lai, L.-G. Chen, and H. H. Chen, "Hardware-efficient belief propagation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 5, pp. 525–537, 2011.

[3] C.-C. Cheng, C.-K. Liang, Y.-C. Lai, H. H. Chen, and L.-G. Chen, "Fast belief propagation process element for high-quality stereo estimation," in *IEEE International Conference on Acoustics, Speech and Signal Processing, 2009. ICASSP 2009.* IEEE, 2009, pp. 745–748.

[4] H. Eslami, T. Kasampalis, and M. Kotsifakou, "A GPU Implementation of Tiled Belief Propagation on Markov Random Fields," in *Proceedings of the Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE '13. Washington, DC, USA: IEEE Computer Society, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=3041405.3041496 pp. 143–146.

[5] "MEMOCODE Design Contest 2013," http://memocode.irisa.fr/2013/, 2013, [Online; accessed 24-September-2013].

[6] M. F. Tappen and W. T. Freeman, "Comparison of Graph Cuts with Belief Propagation for Stereo, Using Identical MRF Parameters," in *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ser. ICCV '03. Washington, DC, USA: IEEE Computer Society, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=946247.946707 pp. 900–.

[7] NVIDIA, "NVIDIA CUDA C Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2018.

[8] S. Xiao and W.-c. Feng, "Inter-block GPU communication via fast barrier synchronization," in *IEEE International Symposium on Parallel & Distributed Processing, 2010. IPDPS 2010.* IEEE, 2010, pp. 1–12.

[9] "Middlebury Stereo Vision Page," http://vision.middlebury.edu/stereo/eval3/.

[10] T. Taniai, Y. Matsushita, Y. Sato, and T. Naemura, "Continuous 3D Label Stereo Matching Using Local Expansion Moves," in *IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017. TRAMI 2017.* IEEE, 2017, pp. 2725–2739.

[11] C. Olsson, J. Ulen, and Y. Boykov, "In Defense of 3D-Label Stereo," in *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '13. Washington, DC, USA: IEEE Computer Society, 2013. [Online]. Available: https://doi.org/10.1109/CVPR.2013.226 pp. 1730–1737.

[12] V. Kolmogorov and R. Zabih, "What Energy Functions Can Be Minimized via Graph Cuts?" Cornell University, Ithaca, NY, USA, Tech. Rep., 2001.

[13] Y. Boykov, O. Veksler, and R. Zabih, "Fast Approximate Energy Minimization via Graph Cuts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, no. 11, pp. 1222–1239, Nov. 2001. [Online]. Available: https://doi.org/10.1109/34.969114

[14] Y. Boykov and V. Kolmogorov, "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 9, pp. 1124–1137, Sep. 2004. [Online]. Available: https://doi.org/10.1109/TPAMI.2004.60

[15] V. Kolmogorov, P. Monasse, and P. Tan, "Kolmogorov and Zabihs Graph Cuts Stereo Matching Algorithm," *Image Processing On Line*, vol. 4, pp. 220–251, 2014.

[16] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing," *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 28, no. 3, Aug. 2009.

[17] C. Barnes, E. Shechtman, D. B. Goldman, and A. Finkelstein, "The Generalized Patchmatch Correspondence Algorithm," in *Proceedings of the 11th European Conference on Computer Vision Conference on Computer Vision: Part III*, ser. ECCV'10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927006.1927010 pp. 29–43.

[18] C. R. Michael Bleyer and C. Rother, "PatchMatch Stereo - Stereo Matching with Slanted Support Windows," in *Proceedings of the British Machine Vision Conference*. BMVA Press, 2011, http://dx.doi.org/10.5244/C.25.14. pp. 14.1–14.11.

[19] J. Žbontar and Y. LeCun, "Stereo Matching by Training a Convolutional Neural Network to Compare Image Patches," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 2287–2318, Jan. 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2946645.2946710