GPU-accelerated Height Map Estimation with Local Geometry Priors in Large Scenes

Alireza Rezaei, Nicola Pellicanò and Emanuel Aldea
SATIE - CNRS UMR 8029, Paris-Sud University, Paris-Saclay University, France
{alireza.rezaei, nicola.pellicano, emanuel.aldea}@u-psud.fr

Abstract

Detection and tracking of pedestrians in vast crowded areas is a complex problem addressed actively by the computer vision community. Proposed algorithms should ideally tackle issues of accuracy and speed at the same time. Lengthy computation times for high-quality optimization-based algorithms relying on multiple sensors make them impractical to use on long and detailed sequences. Hence, an efficient acceleration scheme, which preserves the overall accuracy, is vital to be considered. In the current work, we iterate various steps taken to accelerate a multi-camera pedestrian detection algorithm formulated as an optimization of a height map with local scene geometry constraints. The work is performed using the NVIDIA CUDA framework which allows us to efficiently utilize GPU processors and optimize the various memory accesses. The final results show more than 1000x speedup on real data frames. With respect to preserving the output accuracy, we achieve an accelerated output which is more than 99.9% *in agreement with the original results.*

1. Introduction

Computer vision has made significant progress in the last decade toward more accurate, more robust and faster processing of the video data deluge that we create and use. Non-crowded scenes have represented for a long time the main area of interest for the computer vision community, and pedestrian detection algorithms evolved significantly in the last decade, addressing complex applications such as identification of people, grouping analysis, estimation of body parts, gesture based and trajectory based action analysis etc. However, as it has been already highlighted many times, all these methods are not appropriate when high-density crowd analysis is performed, and new methods must be designed in order to cope with extreme clutter. While clutter is the main difficulty, practical considerations also raise difficult questions, i.e. the use and topology of a network of sensors, the transfer and the processing of the data, or the data fusion strategies. Technical difficulties widen the gap between proof-of-concept experiments aimed at high-density crowded scenes and functional solutions.

Our work addresses a difficulty which is pervasive in pedestrian detection tasks relying on multiple camera networks with overlapping fields of view, namely the joint detection of pedestrians which project potentially in different sensors. During the last decade, a number of works [9, 1] addressed this matter under various simplificatory assumptions incompatible with strong clutter, which impede their applicability in high density areas. The underlying rationale is that at least an approximate detection at object level must be performed before relating uncertain detections from different views. On the other hand, a low-level fusion strategy, such as the one proposed in [2], is applied *before* the detection step and is expected to cope better with ambiguous scenes specific to crowded environments. Recently, the work of [11] redefined the low-level fusion as a global height map estimation taking into account the local geometry of the scene, represented by the accurate ground plane location and the vertical vanishing line direction (Figure 1). The optimization was solved using a Loopy Belief Propagation algorithm, but the price for a high-quality solution compared to a heuristic based one is the computational cost (around 18 minutes per triplet of synchronized frames).

The objective of the current work is to reformulate the algorithm proposed in [11] and to accelerate it by exploiting a massively parallel architecture, in order to render it compatible with practical applications. Section 2 offers a synthetic overview of the initial algorithm, then in Section 3 we highlight at different levels the various optimizations which were performed in order to adapt the algorithm to the considered architecture. Section 4 presents the results of the acceleration, and then we conclude in Section 5.

2. Algorithm overview

In [11], the authors propose an unsupervised pedestrian detection method based on information derived from mul-

tiple camera sources. They formulate the detector as a Markov Random Field (MRF) based stereo matcher, which has to minimize a global energy by assigning some labels to each pixel p of the reference image $\mathcal{I}$:

$$E(l) = \sum_{p \in \mathcal{I}} D_p(l_p) + \lambda \sum_{(p,q) \in \mathcal{N}} V_{p,q}(l_p, l_q) \qquad (1)$$

where: (i) given the label set $\mathcal{L}$, l is a labeling assigning a value $l_p \in \mathcal{L}$ to each $p \in \mathcal{I}$; (ii) $\mathcal{N}$ is the set of edges of the image graph (4-connectivity is assumed); (iv) D_p is the data cost function; (v) $V_{p,q}$ is the discontinuity cost function; (vi) λ is a regularization parameter. The detection task is performed by the use of a special *unknown* label implying that no pedestrian is present at that pixel.

The estimated scene geometry leads to the ability to project every pixel of one image in another one knowing its height above the ground plane (in meters). Thus, it is possible to bound the desired range of detection for a pedestrian to an interval of heights (from $1.4m$ to $2m$ in their experiments). As a consequence, they define the labels of the energy as height values (as opposed to depth values of classic stereo matching approaches). In terms of computational efficiency, such choice simplifies the problem of labels assignment, since each pixel, at a fixed range of heights, can live in a range of depths which is not fixed and which does not scale linearly.

The data cost is expressed as the combination of DAISY [13] dissimilarities between the reference pixel and the possible correspondences (lying along an epipolar segment) in each of the other views. The data cost is an input and can be precomputed, thus it has no impact when studying the acceleration of the optimization (except the initial access time).

The discontinuity cost derives from the observation that the direction of expected maximum height variation $\mathbf{r}_p$ points towards the vertical vanishing point. Moreover, with the estimated geometry, the magnitude of such variation $|\nabla_p|$ can be estimated. While such value depends also on the current estimated height of l_p,the authors suggest that pre-computing it for an average height leads to a negligible error, and thus this will be exploited as a fixed parameter in the acceleration. Being $d(p, q^{\perp})$ the projection of pixel q av$\mathbf{r}_p$, the discontinuity cost is estimated as [11]:

$$D_{p,q}(l_p, l_q) = \left| l_p - l_q - s_p |\nabla_p| d(p, q^{\perp}) \right| \qquad (2)$$

In computational terms, the pre-computation of such cost for any possible pairs of labels is not scalable with the domain cardinality, and thus should be avoided. However, the term $s_p|\nabla_p|d(p, q^{\perp})$ could in theory be precomputed, since each node p relates to 4 possible values (one for each neighbor), and only 2 of them are distinct in absolute value (while the other two being their negation).

With the data cost, height variation magnitude, and a part of the discontinuity cost being possibly precomputed, the trade-off between computational speed and memory footprint becomes critical for the speedup of the algorithm.

To minimize the energy function (Equation 1), the Loopy Belief Propagation (LBP) algorithm has been used. LBP is an algorithm for computing approximate marginal statistics over graphs with cycle. LBP works over a graph by transmitting *messages* between nodes and computing *beliefs* at each node [12]. Thus, each iteration of the algorithm consists in computing new messages between each neighboring nodes. An outgoing message from m to n is computed by considering all the incoming messages to m except the one coming from n itself. After convergence, the belief for a given node is computed by considering all the incoming messages to that node [8].

3. GPU Implementation

In recent years, general purpose processing on GPU (Graphics Processing Unit) provided significant support for many scientific fields in which efficiency and speed is a vital factor. Specifically, NVIDIA CUDA framework has enabled us to rely on a GPU parallel environment with greater ease. [4] describes the structure of NVIDIA GPUs in two levels: each GPU chip consists of streaming multiprocessors (SM), which have their own cores (Figure 2). CUDA uses the term 'grid of thread blocks', where multiple blocks map onto multiple SMs and multiple threads map onto the cores. Each thread block contains several threads. Each thread has its own local memory while it can also communicate with other threads inside the block via shared memory. Every thread has also access to a bigger and slower global memory. GPU groups threads together in *execution warps*. Threads inside a warp execute concurrently. The size of warps depends on the device being used (in our case it equals to 32). Generally, it is not necessary to follow the exact hardware specifications when utilizing CUDA; we can use more threads than cores and leave the scheduling to the hardware. Having these details in mind, the next section will describe our approach to the GPU implementation.

3.1. Basic optimizations

To optimize the pedestrian detection algorithm, we started with basic changes from a mostly serial CPU implementation to a highly parallel GPU code. The next three sections iterate over the main decisions taken in order to make the translation as efficient as possible.

3.1.1 Thread mapping

The first step to translate a serial algorithm to a parallel paradigm is to assign the responsibility of each processing

(a) left camera (b) central camera, overlaid detections (c) right camera

Figure 1: Regents Park Mosque dataset, *Dense* sequence [11]. Three camera views are used.

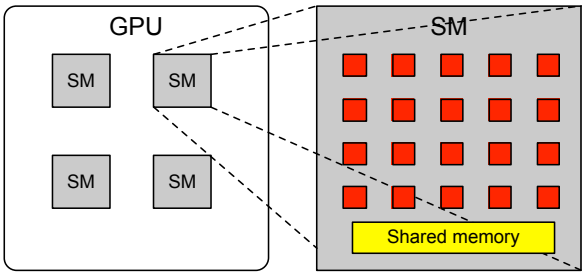

Figure 2: GPU architecture hierarchy (adapted from [4])

unit (in our case GPU threads). In our problem, this means to specify how the grid of threads is related to our computation grids. As our data come from 2D images, if we decide on different dimensions for the thread blocks we need a proper translation between the two. In the remainder of this section N will denote the number of pixels, M the number of messages for each pixel and V number of labels.

First we choose the responsibility of each thread block.

One pixel per thread block: In this configuration, we will have N thread blocks either in 1D fashion or similar to the image in 2D with a (Width,Height) matrix of thread blocks. Each block tends to messages originating from only one pixel. So each thread depending on its block index tends to a different pixel.

Multiple pixels per thread block: This way, we assign more than one pixel for each thread block in hope of doing more work in parallel. Each thread, depending on its block index and also its thread idx (for example third dimension of the thread index) knows which pixel to access.

In practice, putting computation of more than one pixel in a block will not give us any improvement. In our tested GPU we had the limit of 32 active blocks and 64 active warps per SM. This means with just 2 active warps per thread block we can theoretically achieve 100% occupancy for the GPU. In our case, for computing four messages for one pixel we need four active warps (assuming warp size is 32 and $V < 32$). Therefore, there is no actual need for more active warps per thread block.

We also have to arrange threads inside a block.

Each block containing M*V threads: Each thread is in charge of calculating the message for one neighbor regarding one particular label. Whether the formation is in one dimension or two will not affect the performance but structuring the block as M vectors of length V helps the programming process.

Each block containing M*32 threads: This means instead of using the number of labels as the width of the block we use the nearest power of two greater than V. This way we make sure each computation warp only deals with messages related two one neighbor.

As mentioned in [5], GPU architecture follows a Single Instruction Multiple Data (SIMD) execution model, which is not suited for kernels with divergent execution flow. Thus, as a general rule we need to make sure threads in the same warp follow the same (or similar) path. In our case, there is significant divergence between code execution paths for different messages. Therefore, in order to keep the divergence minimal in each warp, we chose the dimensions of the blocks to be M*32 (assuming the warp size is 32), limiting each warp to one message.

After deciding on how to map to the GPU threads, the conversion to parallel code is straightforward, as loop indexes are replaced by thread and block indexes. In the next two sections we describe in more details two major parts of the parallel code.

3.1.2 Parallel reduction

Part of the message passing process used in this algorithm is to compute minimum and sum of messages. Both of these fall under the family of reduction vector operations (i.e. deriving one value from a K-sized vector). Therefore, to parallelize these operations we can follow the same procedure. To choose the best option, we considered two conventional approaches to a parallel reduction mentioned in [7] and [10]. The first approach uses each thread block shared memory to reduce at each step two elements of the vector. This means we will need a vector of size V in shared memory for each reduction. The reduction algorithm made possible on new GPUs [10], uses a process called shuf-

fling to communicate a variable between threads inside a block. This way there is no need for shared memory and synchronized access. After testing in our case, which we uses around 400k blocks and 25 labels, shuffling has been more effective as expected, although the difference is not considerable but significant.

3.1.3 Computing the discontinuity cost function

During each LBP iteration, considerable time is dedicated to the calculation of the discontinuity cost function. This part includes many summations and multiplications which, as mentioned before, are the same at each iteration. Therefore it is a good idea to pre-compute this part and exclude it from the run-time and turn it into a single memory look up. Each run of the function needs six arguments: two pairs of pixel locations and two labels. Since the locations are always neighbors, we can reduce the input to one pair for a pixel location and another argument indicating the direction of the neighbor. This means for completely pre-computing this function we will need a five dimensional matrix. The size of the discontinuity matrix S_{DM} will be equal to:

$$S_{DM} = H * W * 4 * V^2 \tag{3}$$

where H and W are the input image height and width respectively. Complete pre-computation of the function led to speed up in runtime of each iteration; but at the same time cost more than 4GB of graphics memory, for 400k pixels and a label set of 25. This could prove problematic if we decide to increase the label-set or use larger images. Therefore another approach was used to divide the function in two parts: one was to be computed beforehand and one to be calculated at every iteration. The part of the function related to the geometry is precomputed and stored in a matrix with a size proportional to the size of the image. The computation depending on the labels proved to be very simple and manageable in each iteration. This way we decreased the amount of required memory to around 1.5GB which is a fairly reasonable usage. This new approach also decreased the number of global memory accesses. Before we had V^2 global memory access per message. Now we have only one per message. This gave us an overall improvement on iteration runtime.

3.2. Further optimization

In this part, we will briefly iterate over some further optimizations done in order to run the algorithm as efficiently as possible. Some of these points are general best practices which will work on any parallel GPU program, but in some cases the changes make sense only in the context of our problem.

3.2.1 Memory optimization

In the process of minimizing the mentioned energy function, there are three major group of stored data: pre-computed data cost function, partially pre-computed discontinuity cost function and previously calculated messages. All these sources are needed for computing new messages, therefore in each iteration there are many memory reads from the GPU global memory. Also, in the final step of the computation we need to store the messages in the global memory again to be used in the next iteration. This makes a calculated memory access scheme vital for the algorithm to run as fast as possible. We will mention two general important points to achieve a better access time.

Benefiting from memory hierarchy: In some cases we need to use the same data more than once. The data cost function in our case is frequently needed. In these cases, it is not advisable to access the global memory each time. The best choice is to load the data once and store it in a faster memory, either shared memory or each threads local memory. The choice between the two depends how much of each memory we have available to use. Overuse of local memory can lead to using more registers which can reduce the overall occupancy of GPU. For example, in our tested GPU each block was limited to 65536 registers and 98KB of shared memory.

Coalesced access: [6] considers coalescing memory operations as one of the general optimization directives for a parallel program which can lead up to 10x speedup. In simple words, this means consecutive threads access consecutive memory addresses (among other conditions). When accessing incoming messages to each pixel it is not possible to maintain a coalesced access but in other cases maintaining such access gave us considerable improvement.

3.2.2 Instruction optimization

After optimizing the memory operations, the bottleneck of the kernel falls on too many mathematical instructions. Mostly in the discontinuity cost function, floating point division and multiplication caused a considerable delay. One possible solution to alleviate this problem is to sacrifice accuracy for more speed by using GPUs fast mathematical instructions. Actual usefulness of this approach obviously varies case by case. In our algorithm, considerable trial and error with these functions led to use of __fdividef, __fmul_rd and __fsub_rd instead of regular division, multiplication and subtraction in limited cases. This gave us reasonable imprecision which did not affect the end result while saving significant time.

3.2.3 Algorithm optimization

The last step in our optimization process was to investigate whether or not any change in the core algorithm can be helpful. Following the advise of [3], we decided to alternatively calculate only half of the messages in each iteration. Basically, if we divide the pixels into two subsets A and B following a checkerboard pattern, in each iteration we only compute the outgoing messages of pixels belonging to either A or B. This makes sense because the messages sent from nodes in A only depend on outgoing messages of nodes in B. The same can be said about calculation of messages from nodes in B. [3] also describes the new message from node p to q at iteration t as: if t is odd (even) then

$$\bar{m}^t_{p\to q} = \begin{cases} m^t_{p\to q} & if\ p \in A\ (if\ p \in B) \\ m^{t-1}_{p\to q} & otherwise \end{cases} \quad (4)$$

This means the new messages are almost the same as the standard ones, and regarding convergence this method also converges to the same fixed point i.e. after convergence $m^{t-1}_{p\to q} = m^t_{p\to q}$ While being an approximation, this change practically did not affect the end result of the algorithm but it managed to decrease the iteration time by half.

4. Results

In this section, we will report the recorded execution time of the algorithm using real captured data on a selected platform. For testing, we have used a system with NVIDIA Geforce GTX1080 graphic card and 8GB of graphical memory. The system is also equipped with an Intel® Core™ i7-6900K CPU with 3.20GHz processing speed. The iteration time was measured using NVIDIA's own profiler *nvprof*. Figure 1 shows a sample input/output of the algorithm which is extracted from a sequence of captured images. The images cropped down to the size of 781*621 which will give us around 500K pixels to work with. As mentioned before, the algorithm is tested with a label set of size 25; from 1.4 meters to 1.975 with inclusion of a special label for pixels without heads. The algorithm consists of three main kernels - one for pre-computation, one for each iteration of loopy belief propagation, and one for final belief computation. The first and last kernel will run once for each image while the second one will run depending on the number of iteration needed for convergence, in this case 100. Figure 3, shows the general time-line of the program. As we can see, about 98% of the runtime is occupied by the iteration kernel, which makes it the most important kernel to optimize. Figure 1 shows the execution time of these kernels using ten consecutive frames.

Since more than 98% of execution time belongs to the iteration kernel, we will conduct our comparison with CPU code only for this kernel. We achieve over 3000x speed-up for each iteration. As said earlier, the process of getting to this level of efficiency involved several general purpose optimizations as well as some strategies specific to our problem. Table 2 shows the progression of iteration time for a sample frame in different version of the algorithm, from initial serial code to the current most optimized state.

Frame	precomputation	Average iteration	Belief computation
1	3.854	3.781	3.201
2	4.268	4.267	5.415
3	3.66	4.147	4.016
4	3.788	3.799	3.054
5	3.899	4.401	3.983
6	3.936	4.032	3.203
7	3.886	3.934	3.704
8	3.79	3.8	3.235
9	5.031	4.262	3.116
10	4.284	4.101	3.094
average	4.0396	4.0524	3.6021

Table 1: Time taken by the three main kernels in 10 consecutive frames. Times are in millisecond

Version	Iteration time
CPU code	$\sim$11s
Naive GPU implementation	$\sim$1s
Using initial pre-computation	$\sim$0.25s
Memory optimizations	$\sim$0.05s
Improved pre-computation	$\sim$0.02s
Using fast instruction approximation	$\sim$0.01s
Message passing approximation	$\sim$0.004s

Table 2: Step by step optimizations and their time in seconds

It was crucial to make sure no part of the optimization introduces any intolerable deviation from the original output. Inherently, CPU and GPU codes give us slightly different floating point operations which can cause differences in final labels. On top of that, as covered in section 3.2, we have used two approximations namely the use of fast arithmetic operations and computing alternatively only half of the outgoing messages. Depending on each frame these changes cause some misidentification and misjudgment of height labels. A comparison of both versions of the algorithm was done on a set of 15 consecutive frames. Table 3 shows the result of this comparison. Three metrics are considered: the number of pixels in which both algorithms detect a head but disagree on the height (second column); the number of pixels in which the two implementations disagree on the presence of a head (third column), and finally average of absolute difference for the pixels which disagree on the height. As we can see, considering the total number of pixels, very few misidentifications happen and also the

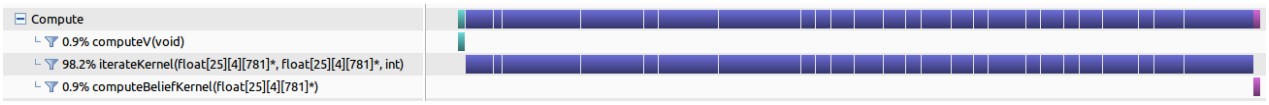

Figure 3: Overall time-line of the execution of belief propagation for one frame with 100 iterations

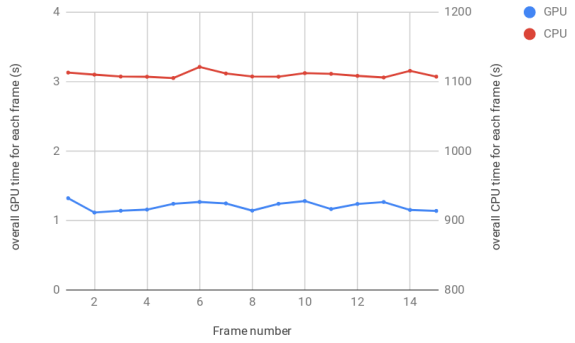

Figure 4: Comparison between the CPU and GPU code in total execution time for each frame

height difference stays around 2.5cm which is the elementary height increment.

Figure 4 shows the overall execution time for each frame in both CPU and GPU code. These times include the reading of precomputed data cost function from disk. Overall, a solid 1100x speedup means we can process a long video sequence which used to take several days, in a matter of minutes.

Frame	Number of pixels with wrong height	Number of pixels with wrong identification of head presence	average absolute difference of height (meter)
1	18	5	0.025
2	44	62	0.025
3	41	12	0.025
4	34	11	0.025
5	31	5	0.025
6	2	3	0.025
7	47	112	0.025
8	14	9	0.025
9	15	3	0.025
10	3	2	0.025
11	41	32	0.032
12	60	15	0.046
13	43	10	0.04
14	50	15	0.027
15	18	1	0.038

Table 3: Result of comparing the output of optimized code with the original CPU code. Window size is 781*621 (485001 pixels)

5. Conclusions

In the current text, we iterated over several steps taken to optimize and accelerate the multiple camera pedestrian head detection algorithm [11] and adapt it to run efficiently on GPU using NVIDIA CUDA framework. We presented the details of the adaptation for the GPU structure as well as problem specific approximations which helped reaching greater speed without losing precision. In the end, we achieved over 1000x speed up in overall execution time for each frame, while preserving over 99.9% of original outputs.

References

[1] A. Alahi, L. Jacques, Y. Boursier, and P. Vandergheynst. Sparsity driven people localization with a heterogeneous network of cameras. *JMIV*, 41(1-2):39–58, 2011.

[2] R. Eshel and Y. Moses. Tracking in a dense crowd using multiple cameras. *IJCV*, 88(1):129–143, 2010.

[3] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. *International journal of computer vision*, 70(1):41–54, 2006.

[4] A. Gray. Best practice guide-GPGPU. 2017.

[5] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. In *Fourth Workshop on General Purpose Processing on GPU*, page 3. ACM, 2011.

[6] M. Harris. Optimizing CUDA. *SC07: High Performance Computing With CUDA*, 2007.

[7] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[8] A. T. Ihler, W. F. John III, and A. S. Willsky. Loopy belief propagation: Convergence and effects of message errors. *JMLR*, 6(May):905–936, 2005.

[9] S. M. Khan and M. Shah. Tracking multiple occluding people by localizing on multiple scene planes. *TPAMI*, 31(3):505–519, 2009.

[10] J. Luitjens. Faster parallel reductions on Kepler, 2014.

[11] N. Pellicanò, E. Aldea, and S. Le Hegarat-Mascle. Geometry-Based Multiple Camera Head Detection in Dense Crowds. In *BMVC - 5th Activity Monitoring by Multiple Distributed Sensing Workshop*, Sept. 2017.

[12] S. C. Tatikonda and M. I. Jordan. Loopy belief propagation and Gibbs measures. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 493–500. Morgan Kaufmann Publishers Inc., 2002.

[13] E. Tola, V. Lepetit, and P. Fua. Daisy: An efficient dense descriptor applied to wide-baseline stereo. *TPAMI*, 32(5):815–830, 2010.