Parallel algorithms to a parallel hardware: Designing vision algorithms for a GPU

Jun-Sik Kim, Myung Hwangbo and Takeo Kanade
Robotics Institute
Carnegie Mellon University
{kimjs,myung,tk}@cs.cmu.edu

Abstract

A GPU becomes an affordable solution for accelerating a slow process on a commercial system. The most of achievements using it for non-rendering problems are the exact re-implementation of existing algorithms designed for a serial CPU. We study about conditions of a good parallel algorithm, and show that it is possible to design an algorithm targeted to a parallel hardware, though it may be useless on a CPU.

The optical flow estimation problem is investigated to show the possibility. In some time-critical applications, it is more important to get results in a limited time than to improve the results. We focus on designing optical flow approximation algorithms tailored for a GPU to get a reasonable result as fast as possible by reformulating the problem as change detection with hypothesis generation using features tracked in advance. Two parallel algorithms are proposed: direct interpolation and testing multiple hypotheses. We discuss implementation issues in the CUDA framework. Both methods are running on a GPU in a near video rate providing reasonable results for the time-critical applications. These GPU-tailored algorithms become useful by running about 240 times faster than the equivalent serial implementations which are too slow to be useful in practice.

1. Introduction

Along with the last fifty years' development of a computer, computer algorithms have been developed enormously. Computers have become faster by increasing clock speed, using bigger and faster memory, and predicting upcoming instructions. Moreover, more cores are put on a single chip to increase the computing capability of a central processing unit (CPU). Based on this progress, recent algorithms can handle more data in a shorter time than before. However, it is still not easy to use the multiple cores on a CPU efficiently, because the most of the algorithms are serial. More importantly, the computation speed is still one of the important issues in spite of the development of both hardware and software.

One noteworthy idea for accelerating processes is to use different hardware architecture. People have paid attention to the parallel architecture of a graphical processing unit (GPU), which has many simple cores. Although each core of a GPU is slower than that of a CPU, many cores on a GPU can run in parallel, and thus, its overall performance can be better than that of the CPU by solving many simple problems with different data concurrently. Utilizing a GPU for non-rendering purpose is called "*General purpose computation on Graphics Processing Units* (GPGPU)".

Nowadays there are many GPGPU implementations in vision research, but most of them are the exact reimplementation of CPU equivalents. Many vision or image processing algorithms for *each pixel* in an image are good to use a GPU. However, when one looks into existing algorithms, there exist many difficulties in implementing them in parallel, mainly because they are designed for a serial CPU. Our claim is that there are better ways for a parallel hardware by changing the formulation of original problems in a parallel way.

In this paper, we consider which algorithm is good or not for a GPU implementation, and show that it is possible to design algorithms which are more efficient on the GPU, but not on the CPU by changing the viewpoint to the original problem. In this paper, we introduce two *GPU-tailored algorithms* for optical flow approximation, which is generally formulated in a serial way.

2. Algorithm good for a GPU: SIMD principle

Parallel hardware including a GPU usually achieves *data level parallelism* based on the "Single Instruction Multiple Data" (SIMD) principle, which means that all the cores do the same with different data. To ensure this SIMD principle, an algorithm should not have data dependency between

threads, which forces the algorithm to be serial. In addition, it is not good to have internal branches depending on data, because different data flow implies different instructions of each core. Another consideration is about accessing data. If multiple threads try to read data on the same location, it may be serialized. In writing data on the same location at the same time, it must be a big problem causing data conflict. One may try to avoid this using a critical section such as a mutex or a semaphore as in accessing shared data in a CPU, but it makes the threads stand on a line to read/write data, which breaks the SIMD principle. For example, simple accumulation is not easy on the GPU. If one tries to find an intensity histogram from an image, this seems to be very easy; for each pixel, read its intensity and increase the counter corresponding to the intensity. However, this does not work. Multiple threads may read the same counter, add one, and write it to the same location simultaneously, which results in a wrong result. This accumulation is a very serial process.

Thus, a good algorithm for a GPU is that each thread
1) has memory storage exclusively accessed,
2) does not use the result of the other threads, and
3) does not branch internally depending on data.

In developing algorithms, keeping these conditions are very hard. The NVIDIA CUDA framework alleviates this hardness by making various memory spaces with different R/W characteristics, and by using thread blocks which has multiple threads to communicate each other.

In the rest of this paper, we study optical flow estimation algorithms, which are generally based on the smoothness constraint. It breaks the data independency between threads by relating neighboring pixels' results to each other.

3. Given problem: Optical flow estimation

Estimating optical flows from an image sequence has been investigated for a long time, because the optical flow is one of the most important features used in various applications such as motion estimation, video compression and three dimensional reconstructions. However, it is not easy to solve, because it is an *ill-posed* problem [4], that is, there are an infinite number of possible solutions mathematically. Thus, the problem should be solved by minimizing a predefined cost function considering physical constraints such as a smoothness constraint and an ordering constraint [6]. Estimating optical flows of all pixels in images becomes a very large optimization problem.

To solve the problem in a limited time, one possible approach is to give more strict constraints so that the solution space can shrink to reasonable size. Gradient based image alignment algorithms take this approach [8, 3]. They formulate the problem as a local optimization minimizing intensity differences between local image patches of two input images. By adopting Newton-type optimization method, the optical flows can be estimated within a pixel. To obtain larger flows, coarse-to-fine approach is applied by making pyramid images of the input images. One problem in this approach is the selection of the patch size. Because the image alignment method minimizes the intensity difference, it can not be applied to homogeneous image region. Thus, the patch size is sufficiently large so that there are image variations in the patch. If the patch size becomes larger, the computation takes longer. The image alignment method takes quite long, and is not good for images containing large homogeneous regions in the images.

Another approach is to formulate the problem as a combinatorial optimization. Although a combinatorial optimization is NP-hard and can not be solved in a limited time, it is possible to find a sub-optimal solution by using Markov random field (MRF) model and practical algorithms such as a graph cut or a belief propagation [5, 12, 14]. Even though the practical algorithms give estimation results within a limited time, it still takes long in practice.

In some applications, it is not required to have very accurate dense optical flows, though. For example, obstacle avoidance or local path planning of a mobile robot can be done with rough estimation of the dense optical flows, but the time constraint is more critical to make the robot moving continuously [2, 9].

What we would like to do is to get dense optical flows from two input images on a moving vehicle. The dense flows may have very large disparities and discontinuities. This is a challenging problem because the large disparity gives larger search range, and usually takes a long time. The discontinuity also makes the problem harder because it requires to solve a large sized optimization.

We aim to design parallel algorithms to *approximate* optical flows between images as fast as possible, rather than to estimate them accurately. At first, we reformulate the problem as a *change detection* problem with hypothesis generation from available information such as features tracked in advance, and data from an inertial measurement unit (IMU). Hypotheses are generated with the tracked features and checked whether they are acceptable or not. This formulation makes the algorithms have only per-pixel operations which a GPU can deal with efficiently in parallel.

The proposed algorithms are extremely simple, but out of considerations in the serial implementation because of its inefficiency. However, they are efficient enough on the GPU because they are *designed* considering the SIMD principle for the GPU.

4. Approximation of optical flows

In developing optical flow approximation algorithms, we assume that features tracked between images are available. The matched features can be estimated by various methods such as a KLT tracker [8], SIFT matching [7], or even a

simple template matching. If available, rotation information from an IMU or a gyroscope is beneficial.

We observe two important tendencies of the optical flows: *spatial coherency* and *color coherency*. The spatial coherency means that spatially near pixels tend to have similar optical flows. The color coherency is that pixels having similar colors tend to have similar flows. These are based on the quite aggressive assumption that the pixels in the *same object* have similar colors as well as the similar flows. Using these properties, we will design algorithms for each pixel without any dependency between pixels.

4.1. Method 1: Direct interpolation

We assume that the tracked features are good samples of the input images and the flows that we approximate. Based on this assumption, we can formulate the problem as a maximum likelihood estimation (MLE) problem.

When the color of a pixel p on the position (x,y) is (r,g,b), and a tracked feature point f_i on (x_i,y_i) in the image I_0 has the color (r_i,g_i,b_i), the likelihood $L(p,f_i)$ between them is defined

$$L(p,f_i) = N(\mathbf{v}-\mathbf{v}_i^f), \tag{1}$$

where $\mathbf{v}$ and $\mathbf{v}_i^f$ are 5-vectors consisting of the position and color of the pixel and the tracked features, respectively. N is a likelihood function and we assume that it follows a 5-dimensional normal distribution whose five variables are independent in order to make the evaluation as simple as possible. Thus, Eq. (1) can be rewritten as

$$L(p,f_i) = \\ N_c(|r-r_i|)N_c(|g-g_i|)N_c(|b-b_i|)N_s(|x-x_i|)N_s(|y-y_i|) \tag{2}$$

where N_c and N_s are 1-dimensional normal distribution functions for color and spatial information, respectively. By using the 1-dimensional standard normal distribution N_1, the likelihood function $L(p,f_i)$ becomes

$$\begin{aligned} L(p,f_i;\alpha,\beta) = \\ & N_1(\alpha|r-r_i|)N_1(\alpha|g-g_i|)N_1(\alpha|b-b_i|)\cdots \\ & N_1(\beta|x-x_i|)N_1(\beta|y-y_i|). \end{aligned} \tag{3}$$

Now, the likelihood function has two parameters α and β. In practice, these parameters control the smoothness of the object boundary in the approximated optical flow map. If α/β is large, the approximation becomes smoother. We experimentally choose $\alpha/\beta = 4$.

The MLE of the optical flow (u,v) of the pixel p can be calculated easily [10] by weighted averaging of the optical flows (u_i,v_i) of the tracked features f as

$$u(p) = \sum \frac{N(\mathbf{v}-\mathbf{v}_i^f)u_i}{N(\mathbf{v}-\mathbf{v}_i^f)}, \text{ and } v(p) = \sum \frac{N(\mathbf{v}-\mathbf{v}_i^f)v_i}{N(\mathbf{v}-\mathbf{v}_i^f)}. \tag{4}$$

Algorithm 1 Direct interpolation algorithm

Require: Tracked feature points f_i: Position (x_i,y_i), color (r_i,g_i,b_i) and flow (u_i,v_i), and two images I_0 and I_1.

For each pixel p: position (x,y), and color (r,g,b),

1: Calculate the distance $|\mathbf{v}-\mathbf{v}_i^f|$ for every tracked feature f_i.
2: Estimate the MLE of the optical flows with Eq.(4).
3: Test the MLE hypothesis with the second image I_1.

Once the MLE of the optical flows is obtained, it is tested on the second image I_1. We test it as *change detection.* If the pixel color of p' in I_1 considering the flows is not changed from the original pixel p in I_0, the flow hypothesis is accepted. If not, we declined the hypothesis and mark the pixel as "unclassified." The decision is made using a simple threshold.

Although this algorithm is extremely simple, it requires many repetitive multiplications and additions for every pixel, which takes a long time in serial implementation. The overall time complexity is $O(nm)$ with n pixels and m tracked features. However, because the computation depends only on the property of the pixel, we can make n to be n/n_t where n_t is the number of parallel threads running concurrently. Because of the simplicity of the algorithm, it is possible to have a large n_t. We will discuss the implementation issues in Sec. 5.1.

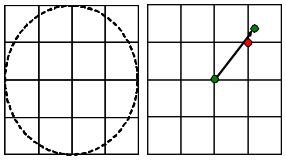
4.2. Method 2: Multi-hypotheses and testing

In the direct interpolation algorithm, we assume that the tracked features are good samples of the color and flow distribution. However, we can ensure neither the goodness of the samples nor the quality of the final approximation. What we can find is only the number of the *unclassified* pixels for the quality measure.

Because we check only one hypothesis from the 5-dimensional likelihood function, if the features are not good samples, the algorithm fails to find the good approximation. To resolve this problem, we test more hypotheses rather than the most likely one.

We build a hypothesis space of optical flows by discretizing the possible optical flows in pixels. The problem is converted to find the best element in this *discretized flow space* (DFS), and it becomes a labeling problem from the given evidences such as a position, colors and flows of the tracked features.

Now, we should evaluate every optical flow hypothesis for each pixel. Thus, the likelihood function becomes a 7-dimensional Normal distribution including the 2-dimensional optical flow vector (u,v). Consequently, the feature vector $\mathbf{v}$ of each hypothesis has (x,y,r,g,b,u,v), two



(a) (b)

Figure 1. Discretized flow space. (a): the flow vector bounds (in dashed line) and its DFS; (b): flows are disctretized to the nearest grid points.

Algorithm 2 Multi-hypotheses and test algorithm

Require: Tracked feature points f_i: Position (x_i, y_i), color (r_i, g_i, b_i) and flow (u_i, v_i), and two images I_0 and I_1.

1: Find the maximums and the minimums of the flows u and v, respectively.
2: Build the *discretized flow space* from the maximum and minimum values of u and v.

For each pixel p: position (x, y), and color (r, g, b),
3: Evaluate the likelihood of the hypotheses in DFS using 7-dim likelihood function using every tracked feature f_i.
4: Sort the hypotheses by their likelihood.
5: Test the n best hypotheses with the second image I_1 from the most likely one.

for a position, three for a color, and two for a flow. The 7-dimensional likelihood function is simply generalized from a 5-dimensional likelihood function of the spatial coherency (3).

To set the DFS between two images, we assume that all the flows are within the bounds of the flows of tracked features. The minimum and maximum of u_i and v_i of the tracked features defines the pixel-level DFS. For example, when the flow u in x-direction swings from -4 to 4, and v in y-direction from -2 to 2, the DFS has 45 (= 9×5) labels. Figure 1 shows the concept of the DFS and how to discretize the flows.

We use a probabilistic voting scheme to evaluate the hypotheses with the given tracked features. Every tracked feature votes for each hypothesis by its likelihood between them. The score of the hypothesis is the sum of the likelihood values from all the tracked features. This is an evaluation of the posterior probabilities of the hypothesis in top-down manner, while the direct interpolation is a bottom-up approach [10].

After evaluating every possible label in the DFS, the labels are sorted by its likelihood, and tested from the most likely one. Once one hypothesis is accepted, hypotheses less likely than the accepted one are not tested. Algorithm 2 shows this process.

Compared to the direct interpolation method, this algorithm has much more computations for evaluating every possible hypothesis. If there are h labels in DFS, the time complexity becomes $O(hnm)$ rather than $O(nm)$ of the direct interpolation. However, the proposed algorithm still depends only on the pixel itself, and is very parallel so that it is good for implementing on a GPU. The implementation details are discussed in Sec. 5.2.

5. CUDA Implementation details

We use the CUDA library [1] to use a GPU for our implementation, because it is easy to write codes and it offers good features to utilize such as a shared memory and a constant memory to be read by many threads simultaneously. In this section, we explain the efficient CUDA implementation in detail, focusing on the data access patterns which is important in the CUDA framework.

5.1. Direct interpolation

In implementing the direct interpolation, there are two kinds of memory accesses: reading/writing the image data and reading the feature data. A pixel color should be read and the calculated flows should be written for each pixel. Because the proposed algorithm is independent on the other pixels, the read and write for each pixel should be done only once. However, the feature data are used in approximating the optical flow of every pixel, and thus, we should access to each feature data n times for n pixels. Furthermore, it can not be coalesced because every pixel should access the same data at the same time.

We use the constant memory space to store the feature data. Because the feature data are accessed repeatedly, we can expect a very high cache hit rate, and the overall performance becomes better. If the amount of the feature data is less than 8KB, the access is as fast as registers. We can use the constant memory because the feature data do not change between a pair of images and all the pixels should see the same feature data.

For the image, we use 3 channel color data, and it makes the coalesced access to the image hard. One of the requirements for the coalesced data access is the alignment of the data by 32 bits. To make it coalesced, we can use the shared memory as an input/output buffer.It is known that this "forced" coalescing is faster than using the texture memory to access the non-aligned data. Using a texture memory is also convenient.

Finally, we can consider using the LUT for the Normal distribution function because it should be evaluated repeatedly. We evaluate it using the GPU, because our experiment tells us that direct calculation is faster than accessing the LUT memory in the constant memory or in the texture memory. This depends on the complexity of computations

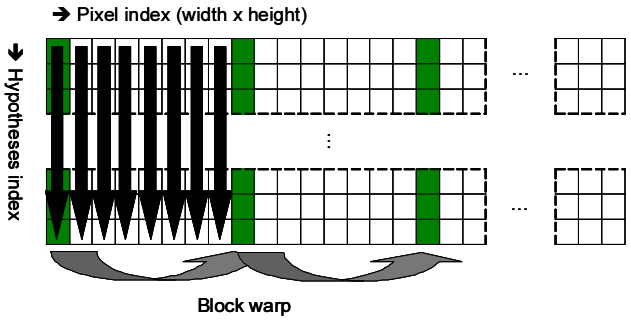


Figure 2. Design of the global accumulator for multiple hypotheses. Note that the adjacent data are accessed by adjacent threads.

in the LUT.

5.2. Testing multiple hypotheses

To deal with multiple hypotheses and their evaluations, we should access a much larger memory space than the direct interpolation method. The major bottleneck of this method is evaluating all the hypotheses for all the pixels. We should configure the global memory space so as to make sure the coalesced access pattern.

Figure 2 shows the design of the global accumulator for the multiple hypotheses. If it is implemented in the conventional CPU in serial manner, it is better to use the transpose of the memory scheme to expect the high cache hit rate because the access to the consecutive memory is much faster and in the serial programming model, the access order is the multiple hypotheses of the same pixel first. However, in the GPU parallelized processing, each thread takes care of each pixel, and transposing the memory map of the accumulator ensures the coalesced memory access pattern.

Finally, we discuss the process flow of the Algorithm 2. The major bottleneck is the evaluation of the every hypothesis for each pixels, and it can easily implemented running on a GPU by using the accumulator design in Figure 2. However, the process has sorting data and, this is not easy to implement in each thread because it should have inner loops and branches depending on data. In addition, complex algorithms like sorting and finding medians, mostly not single-path, tend to use more registers so that the number of the concurrent threads would be reduced. Thus, it is better to sort the hypotheses in the CPU rather than in the GPU, which makes an additional data transfer between the CPU and the GPU, or to make the algorithm have a single path.

To make the process have a single path, we convert the algorithm to find the maximum score values by testing the changeness in advance. The algorithm has now only a single path which is good for a GPU, shown in Algorithm 3. This algorithm is a little different to the original algorithm because this gives the results even if the accepted one has very low score value. However our experiment shows that

Algorithm 3 Multi-hypotheses and test algorithm in GPU

Require: Tracked feature points f_i: Position (x_i, y_i), color (r_i, g_i, b_i) and flow (u_i, v_i), and two images I_0 and I_1.

1: Find the maximums and the minimums of the flows u and v, respectively.
2: Build the *discretized flow space* from the maximum and minimum values of u and v.

For each pixel p: position (x, y), and color (r, g, b),
3: Evaluate the likelihood of the hypotheses in DFS using 7-dim likelihood function using every tracked feature f_i.
4: Test the hypotheses with the second image I_1 if it is changed or not.
5: Find the best hypothesis among the hypotheses which are identified "unchanged".

Figure 3. Two input images and tracked features

there are only few pixels that have the different results with the original one. By doing this, we can get the result much faster because the whole process run in the GPU without the unnecessary data transfer.

6. Experiments

In this section, we show the results of the optical flow approximation from the tracked features, and how fast it is to get the results. For comparison, we implemented the same algorithm both in serial and in parallel using a GPU on the system with the Intel Xeon 3.2Ghz CPU and the NVIDIA 8800 GTX graphics card.

6.1. Performance

Figure 3 shows the input images and the tracked features using a conventional KLT tracker. In this case, the image resolution is 320×240 and the number of the tracked features is 200. We chose these two images to compare the results of the two proposed methods directly, because there is no feature on the pole in the left side of the image, which violates the assumption of the "goodness of the tracked features."

Figure 4 shows the approximation results using the direct interpolation algorithm. The left figure in Figure 4 is

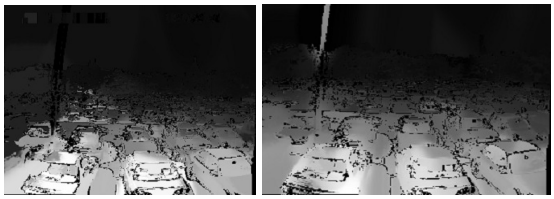


Figure 4. Approximated optical flows by interpolating directly. The pole on the left side is not detected when there is no point on it, while it is detected very well with just one feature.

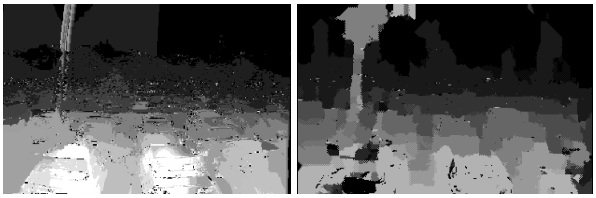


Figure 5. Approximated optical flows by testing multiple hypotheses (Left) and the results by the belief propagation method [5] (Right)

obtained by using the given 156 tracked features. For most part of the image, it approximates the optical flows well, but can not estimate the flows of the pole, because there is no sample features on the pole. When one more tracked feature on the pole is given manually, we can obtain the result in the right image of the Figure 4, which recovers the flows of the pole. This shows that the assumption of the good samples of the tracked features is critical to the performance of the direct interpolation method. However, it is hard to find how good the tracked features are before getting the results.

The serial implementation in C++ takes from 2000 to 3000 milliseconds per frame depending on the 320×240 images with about 150 features. On the NVIDIA 8800 GTX GPU, this takes about 6.2 milliseconds including transferring all the data between the CPU and the GPU. Thus, this parallel algorithm tailored for the parallelized hardware acclerates the execution at least 300 times. Because the KLT tracker using a GPU takes less than 10 milliseconds [11, 13], the overall processing for this interpolation can be done within a video rate.

Figure 5 shows the results of testing multiple hypotheses. Note that this result does not use the added feature on the pole. Even though there are no features on the pole, the proposed method recovers the flows of the pole region accurately. In this case, the number of possible labels in DFS is eight obtained from the tracked features. The estimation result using the belief propagation is shown in the right figure for comparison.

The purely serial implementation takes about 11800 milliseconds. When we implemented the Algorithm 2, which means that only the evaluation is done on the GPU and the sorting and the checking changeness is on the CPU, the evaluation takes about 50 milliseconds and the later serial

Figure 6. Two input images in the "merry-go-round" sequence and tracked features

processing takes 110 milliseconds. For overall process including data transfer between the CPU and the GPU, it takes about 160 milliseconds, which gives 73 times acceleration. Although this is not bad for the optical flow approximation for images with a very large flow vectors, we can make it much faster by using the Algorithm 3 purely running on the GPU. By using the Algorithm 3 purely running on the GPU, we can get the result in 48 milliseconds per frame, which is 246 times acceleration from the pure serial implementation. Even though this is not in the video rate, this gives more reliable results with some bad samples of the tracked features. For comparison, the belief propagation method [5] on the GPU takes 282 msec with 30 message passing stages until convergence to have the result in the right figure in Figure 5.

One can notice that the time consumption in purely parallel implementation is less than the GPU part in the hybrid implementation. That is because there is additional data arrangement for faster processing for sorting on the CPU in the hybrid implementation. Note that the less memory access/transfer gives more acceleration.

We applied out approximation algorithm to the "merry-go-round" sequences. In this case, because of the large motion, the KLT tracking can not be applied. We matched SIFT features [7] between them and refined the results using a robust method. Figure 6 shows the input images and the tracked features.

The number of tracked features is 422 in this case. Figure 7(a) shows the direct interpolation results. Because of the many narrow poles in the images, this sequence is very challenging in finding optical flows between images. For the large areas, the interpolation method gives reasonable results, but it fails to find the flows on the narrow poles. The direct interpolation takes about 14 milliseconds to get the results because the number of tracked features is more than the previous one.

On the other hand, testing multiple hypotheses algorithm generates more reliable results in Figure 7(b). Because the maximum and the minimum values of the flows are quite accurate, the DFS built from the information is a good hypotheses space. In this case, the number of labels is 25, and it takes about 120 milliseconds to approximate it. Figure

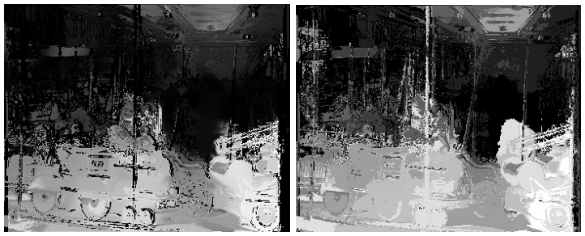


Figure 7. Approximated optical flows by (a) interpolating directly and (b) testing multiple hypotheses

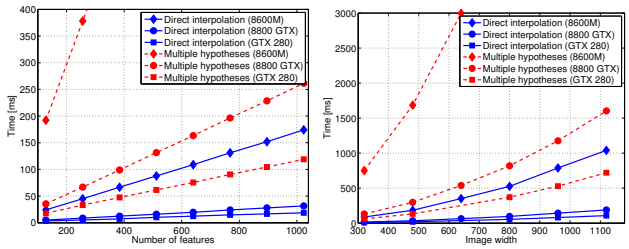
Figure 8. Approximated optical flows by testing multiple hypotheses [14]

8 shows the result by Zitnick *et al*. [14] based on superpixel segmentation. Our implementation on a GPU is less accurate than this state-of-the-art work, but still comparable with much less computation time than globally optimized approaches.

6.2. Time complexity

We analyzed the time complexities of the proposed algorithms with respect to the number of tracked features used, and the image size. As stated in Section 4, the algorithms have linear time complexities to the number of features, because every feature should take part in evaluating every hypothesis. Figure 9(a) shows the time consumption, and it is linear as we expected. Note that the slope of the Algorithm 3 is larger than that of the Algorithm 1, because the former evaluates multiple hypotheses in 7-dimensional space, while the latter does it once in 5-dimensional space. For this experiment, we used 320 $\times$ 240 images with 8 labels for the Algorithm 3, and make 100 trials to measure the time consumed, including data transfers between the CPU and the GPU. The performance is greatly degraded on the NVIDIA 8600M comparing to the others. Because the 8600M video processor has fewer parallel processing cores than the others, the number of parallel threads becomes much smaller.

Figure 9(b) shows the time complexities of the two algorithms to the size of the two input images. In this case, we fixed the number of features to be 512. Because the aspect ratio of images is fixed, the number of pixels increases quadratically. As expected, the algorithms' time complexities increases by the number of the pixels evaluated. In Fig. 9(b), one can notice that the direct interpolation method on the 8600M is even slower than the multiple hypotheses



(a) (b)

Figure 9. Time complexities of the proposed algorithm (a) to the number of features on the 320×240 images and (b) to the image size with 512 features. Blue solid lines denote the direct interpolation method, and red dashed lines do the multiple hypotheses method.

method on the GTX 280. This shows that the performance can be dramatically improved by using more powerful hardware. One can notice that the performance improvement on the GTX280 is not significant compared to the 8800GTX, while it is much bigger between 8600M and 8800GTX. This is because the number of core processors in the 8600M is so small that much more parallel thread block should be generated than the others. The acceleration is made mainly by increasing number of concurrent threads.

7. Conclusions

A GPU is a parallel hardware which can run the same instructions on the multiple data simultaneously. In the traditional approach to use it, existing serial algorithms is ported in the exactly same way to the parallel one, and this is difficult for some algorithms and may be inefficient. It is possible to design parallel algorithms which run on a GPU efficiently, even though their serial equivalent may not.

The optical flow estimation is investigated to show this possibility. Because some applications such as obstacle avoidance or local path planning do not need a very accurate optical flow map, we simply reformulate the optical flow estimation problem as change detection between two images and interpolation using the tracked features in advance. This formulation enables us to deal with the problem as a pixel processing, which is more suitable for a parallel hardware.

We propose two methods: direct interpolation and testing multi-hypotheses. In the direct interpolation, the maximum likelihood estimation of optical flows using the pre-tracked features is tested in the manner of change detection. This method gives a smooth flow map very fast, but if the tracked features do not represent the distribution of the optical flows, there would be many failure in approximating the flows. On the other hand, by testing multi-hypotheses from the pre-tracked features, it is more probable to find more accurate flows, even with the bad representation of the feature

distribution. It takes much longer than the direct interpolation method, but runs in a near video-rate.

Because the methods are approximating the optical flows designed for a parallel hardware, its performance is neither accurate as the state-of-the-art methods, which takes a long time to compute it, nor fast in implementing in a serial manner. However, when we use a parallel hardware such as a GPU, it gives a reasonable result fast enough for the applications which do not require the very accurate optical flow estimations.

References

[1] http://www.nvidia.com/cuda/.

[2] N. Ancona. A fast obstacle detection method based on optical flow. In *European Conference on Computer Vision*, pages 267–271, 1992.

[3] S. Baker and I. Matthews. Lucas-kanade 20 years on: A unifying framework. *International Journal of Computer Vision*, 56(3):221 – 255, March 2004.

[4] M. Bertero, T. A. Pogcio, and V. Torre. Ill-posed problems in early vision. In *Proceedings of the IEEE*, pages 869–889, 1988.

[5] P. Felzenszwalb and D. Huttenlocher. Efficient belief propagation for early vision. *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, 1:I–261–I–268 Vol.1, June-2 July 2004.

[6] B. K. Horn and B. G. Schunck. Determining optical flow. Technical report, Cambridge, MA, USA, 1980.

[7] D. G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, pages 1150–1157, 1999.

[8] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision (darpa). In *Proceedings of the 1981 DARPA Image Understanding Workshop*, pages 121–130, April 1981.

[9] N. Ohnishi and A. Imiya. Dominant plane detection from optical flow for robot navigation. *Pattern Recogn. Lett.*, 27(9):1009–1021, 2006.

[10] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. Mc-Graw Hill, 1984.

[11] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, page published online, November 2001.

[12] Q. Yang, L. Wang, and R. Yang. Real-time global stereo matching using hierarchical belief propagation. page III:989, 2006.

[13] C. Zach, D. Gallup, and J.-M. Frahm. Fast gain-adaptive klt tracking on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPR Workshops 2008. IEEE Computer Society Conference on*, pages 1–7, June 2008.

[14] C. Zitnick, N. Jojic, and S. Kang. Consistent segmentation for optical flow estimation. In *International Conference on Computer Vision*, pages II: 1308–1315, 2005.