# Dense matching GPU implementation

*Author: Hailong Fu. Supervisor: Prof. Dr.-Ing. Norbert Haala, Dipl. -Ing. Mathias Rothermel. Universität Stuttgart*

## 1. Introduction

Correspondence problem is an important issue in computer vision and photogrammetry, which is to look for corresponding pixels across images. Such problem is called image matching. Dense matching is targeting dense and homogenous matching of every pixel; it is an essential part of 3D point cloud generation.

Semi Global Matching (SGM) (Hirschmueller, 2008) is one of the best dense matching algorithms regarding quality and speed. SGM is simpler than global methods, but still computational intensive compared with local matching methods, which challenges its real-time application and fast aerial images processing. However, one good feature of SGM is its high parallelism, which opens the door to GPU implementations.

Meanwhile, CUDA (Compute Unified Device Architecture) (NVIDIA, 2012a) is providing its own GPU multi-threading model using Grid, Block and Thread, which hide much hardware details and is easy to learn. This thesis is targeting the CUDA solution of SGM.

## 2. CUDA

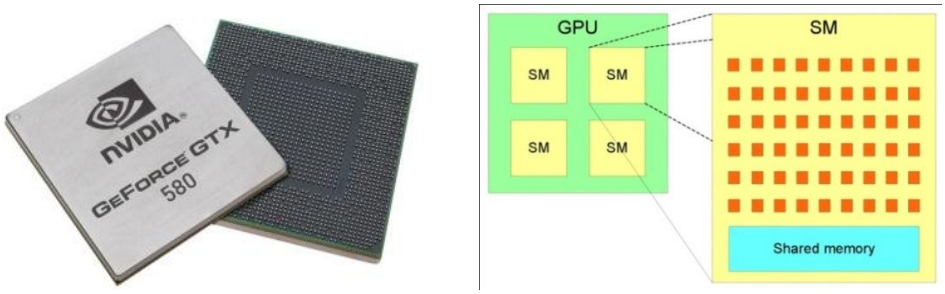



**Figure 1 GPU chip architecture**

All CUDA instructions are carried out in the chip of graphics card. One chip has several Streaming Multiprocessors (SMs) and One SM has several Streaming Processors (SPs). The concepts Grid, Block and Thread are mapped to these hardware components.

Grid means “a grid of blocks”, one grid is executed in one GPU chip, and the blocks in the grid can be 1, 2 or 3 dimensional. Block means “a block of threads”, one block is executed in one Streaming Multiprocessor (SM), and the threads can also be 1, 2 or 3 dimensional. A good choice of the block size is 16x16. Threads are executed in Streaming Processors (SPs), in one block there could be much more threads than the number of SPs, the maximum number of threads per SM is 1536.

Kernel is a C function defined using the "__global__" declaration specifier. Once a kernel is launched, it will be executed as many threads, each thread has a unique ID. Blocks per grid and threads per block should be specified to the kernel before launching.

In CUDA, "warp" is the smallest execution unit. It contains 32 threads and selects threads in block x direction first, a block is composed of N warps. The threads inside one warp are executed exactly synchronous. So CUDA prefers less branched codes otherwise there will be idle threads.

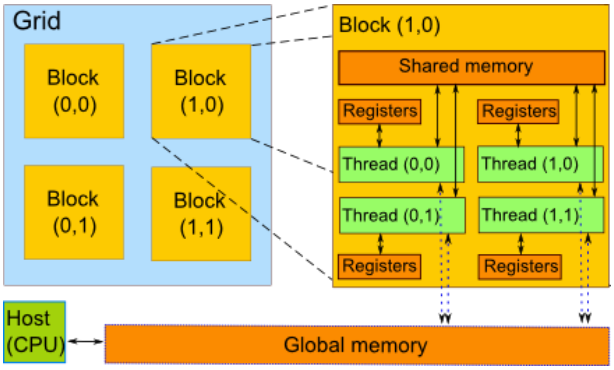



Figure 2 Multi-threading model and memory types

There are different types of memory in CUDA. Their usage should be combined and tested.

Global memory is off-chip and has 1-4GB volume which can be accessed by all threads. Global memory has very high latency and its access is essential for the speed. The most important pattern is called **coalesced access**, which means adjacent threads in one warp should access adjacent memory addresses in the global memory.

Shared memory is on-chip with amount of 48KB per SM. It has very short latency and high bandwidth. If the same address in global memory will be read more than once, shared memory should be used. Shared memory is shared by the threads in one block, but not across blocks.

Registers belong to each thread and can be used to store local variables.

CUDA can also execute multiple kernels concurrently to take full power of device's multiprocessors. Kernels using much shared memory and registers are less likely to be run concurrently.

## 3. CUDA implementation of SGM

Here we assume both base and match images are epipolar rectified, so the matching problem is simplified to 1D, searching can be done in a predefined search range. The matching result is called disparity image in which each disparity means the columns difference of correspondences. The disparity image can be used for triangulation (3D coordinates' calculation).

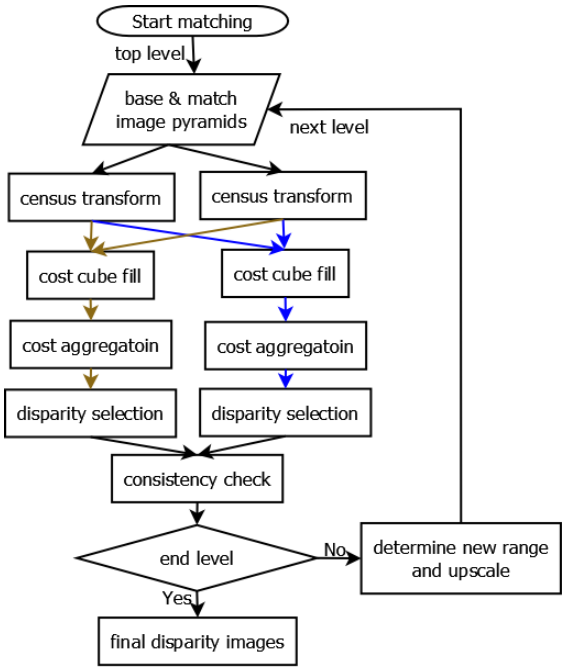



Figure 3 Flow chart of SGM

We are performing the matching in image pyramids, will full disparity range search in the highest level, and dynamic disparity range search in the lower levels.

## 3.1 Census transform

In each level, we first do a census transform for the base and match images to increase robustness. It is done by a 9x7 window visiting each pixel in the image, comparing every pixel in the window and the central pixel, resulting in a 63 bits string containing only 0 and 1. A 2D parallel kernel can be launched to solve this problem. The comparison can be performed on global memory (where images stored) directly, but shared memory is preferred since every pixel will be visited 63 times.

## 3.2 Cost cube fill

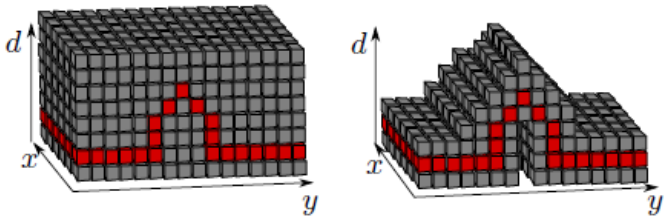



Figure 4 Constant/Dynamic cubes (Rothermel, Wenzel, Fritsch, & Haala, 2012)

Afterwards, for each pixel of the base census image, we calculate a cost value (dissimilarity, e.g. hamming distance) for each disparity. Therefore every pixel can get a string of costs; all cost strings can make a local cost cube. Since disparity range for each pixel is dynamic, the cost cube

is dynamic too. This procedure can be solved by a 3D CUDA kernel, with block x direction pointing to the disparity range (to coalese cost cube access because the cube is stored in the dynamic range first in a linear memory space), block y direction pointing to columns and block z direction pointing to rows.

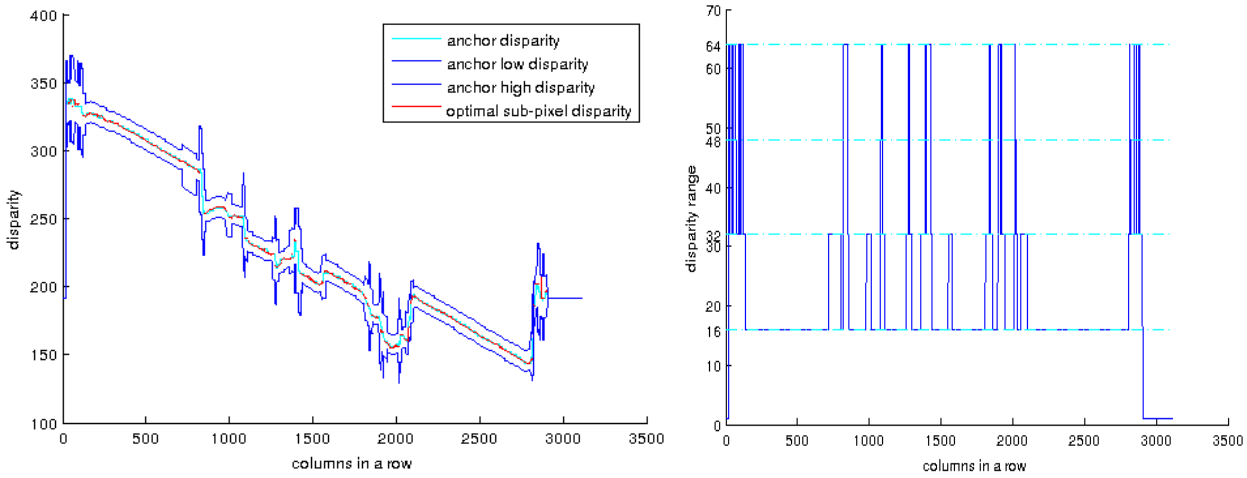



Figure 5 Dynamic disparity range (example)

The selection of block x dimension is dependent on the dynamic range. The maximum range in Figure 5 is 64 while most pixels only have 16. So we chose x dimension 16 but not 64 to avoid idle threads. One thread may process "disparity range/x dimension" steps for each pixel.

## 3.3 Cost aggregation

Cost aggregation is designed for smoothing disparities while preserving disparity discontinuity near edges. It aggregates the local cost cube in 8 or 16 paths.

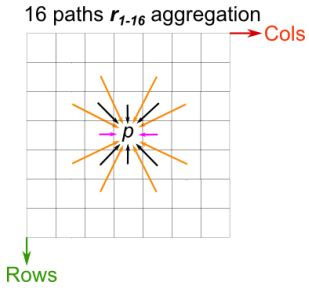



Figure 6 Diagram of 16 aggregation paths

$$L_{\boldsymbol{r}}(\boldsymbol{p},d)=C(\boldsymbol{p},d)+min\left\{\begin{matrix}L_{\boldsymbol{r}}(\boldsymbol{p}-\boldsymbol{r},d)\\ min\begin{Bmatrix}L_{\boldsymbol{r}}(\boldsymbol{p}-\boldsymbol{r},d-1)\\ L_{\boldsymbol{r}}(\boldsymbol{p}-\boldsymbol{r},d+1)\end{Bmatrix}+P_1\\ L_{\boldsymbol{r},\boldsymbol{min}}(\boldsymbol{p}-\boldsymbol{r})+P_2\end{matrix}\right\}-L_{\boldsymbol{r},\boldsymbol{min}}(\boldsymbol{p}-\boldsymbol{r})$$

In one path, this formula shows for every base pixel $\boldsymbol{p}$ and each disparity $d$, how path cost (aggregated) $L_{\boldsymbol{r}}(\boldsymbol{p}, d)$ is calculated from the local cost $C(\boldsymbol{p}, d)$, previous path costs and previous minimum path cost.

The aggregated cost cube (S cube) is a summation over all aggregated path costs and has the same structure as the local cost cube. The optimal disparity corresponds to the minimum S cost.

The aggregation has a recursion behavior because later aggregated path costs are dependent on the earlier path costs. But still it is 2D parallel, for one path, pixels in one row/column can be processed in parallel; the path costs for disparities of one pixel can also be processed in parallel. Therefore we defined aggregation walls to be processed sequentially.

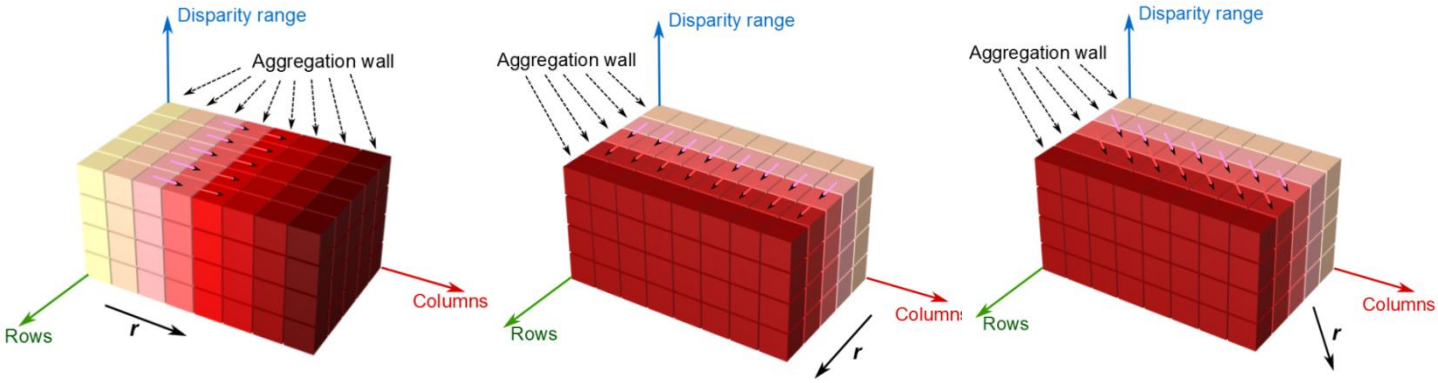



Figure 7 Aggregation walls of horizontal / vertical / diagonal aggregation paths (example)

An aggregation wall is a 2D array of path costs with height equal to the maximum disparity range and width equal to columns/rows. One aggregation wall can be split and processed by many 2D CUDA blocks: x direction points to the disparity range; y direction points to columns. We chose block x dimension to be 16, so one thread may process "disparity range/x dimension" path costs. Each block will read a part of the previous wall and generate a part of the following wall; the width is equal to the y dimension of block. Two walls are allocated for one path; their roles will be switched after once aggregation.

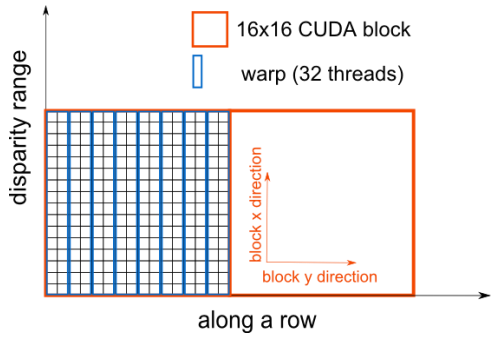



Figure 8 CUDA blocks for vertical aggregation path (example)

To achieve the best speed, we split the 16 paths into 3 groups:

Group one has 6 paths (two vertical paths and four diagonal paths) since their aggregation walls look the same. The 6 path aggregations are performed in 2 passes, each pass for 3 paths (Zhu, Butenuth, & d'Angelo, 2010). This saves local cost cube and S cube access. We store the walls in global memory but not shared memory; otherwise one block cannot access walls diagonally of another block because shared memory belongs to each block. Furthermore synchronization between all blocks of each row is needed due to the diagonal access of walls. So we launch one kernel for one row, another kernel for the next row. When we read the previous wall into the kernel, we can read them into shared memory first because one path cost will be used 3 times.

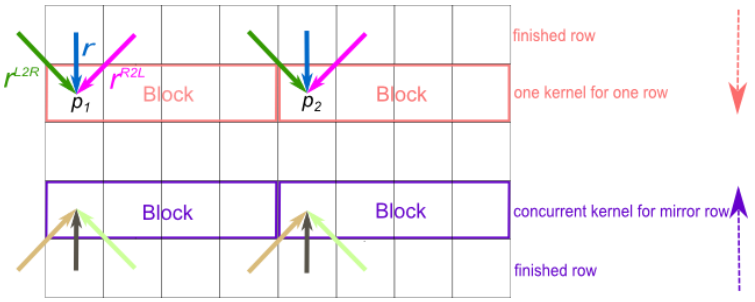



Figure 9 Concurrency of 6 aggregation paths

Group two has 2 horizontal paths. In this case the previous wall and the following wall can always reside and swap in shared memory because they are perfectly aligned.

Group three contains the other 8 inclined diagonal paths, which can be done using the same way of group one. In fact group one and three can be improved by concurrent kernels execution to exploit the peak performance of SMs, due to the independence of up down, down up passes.

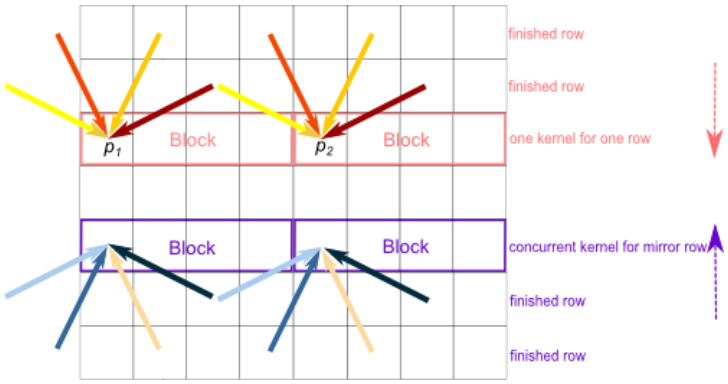



Figure 10 Concurrency of additional 8 paths

The minimum path cost during aggregation can be got by “parallel reduction” (Harris, 2007).

## 3.4 Minimum cost disparity selection

The “parallel reduction” also works well for the minimum S cost disparity selection.

### 3.5 Left right image consistency check

In our program, we implemented a forward-backward matching both from scratching. If the base disparity differs to its corresponding disparity more than 1, then it will be marked as invalid. This can be solved by twice launching a 2D CUDA kernel, in which one thread check for one pixel.

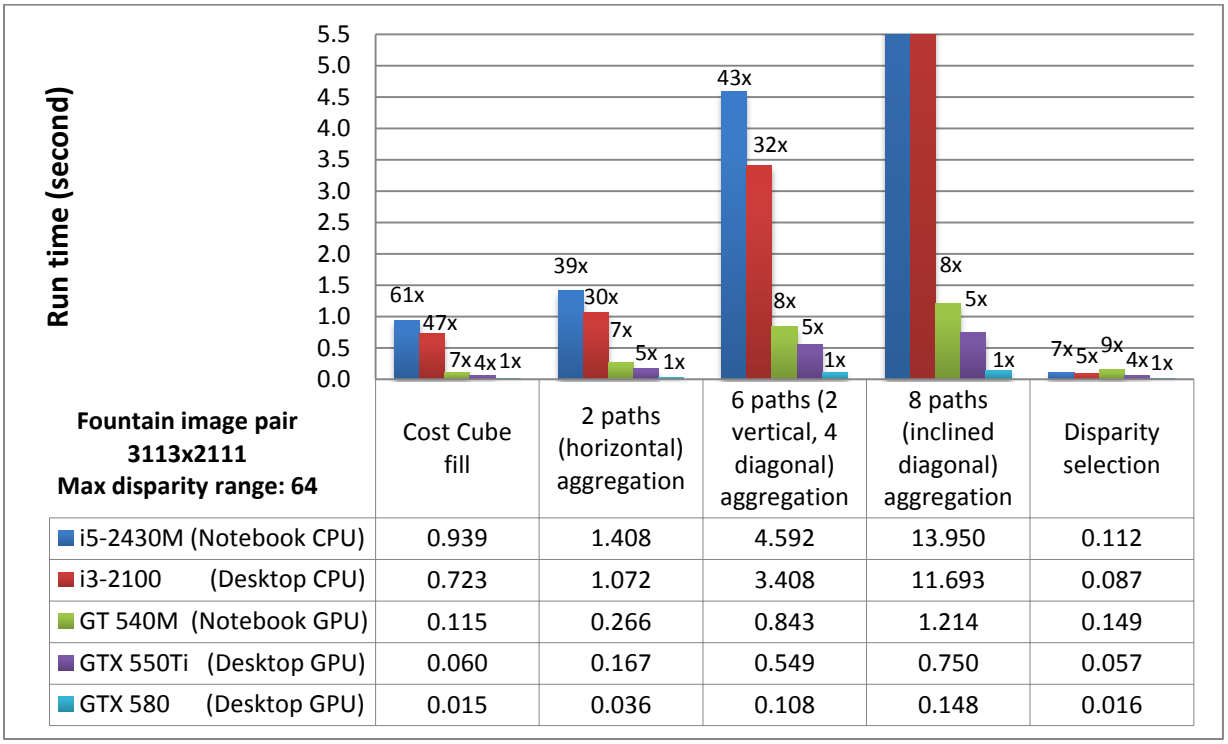
### 3.6 New range determination and upscaling

The new disparity range determination is based on the disparity image from previous pyramid level. We can use a 2D CUDA kernel to visit each pixel, find the maximum and minimum disparities inside a window, and use the difference as the new disparity range. For the pixels marked as invalid, we assign the maximum disparity range. The upscaling of disparity image and disparity range images is just a data copy with doubled size and value. A 2D kernel can be used.

## 4. Results

| | |
|---|---|
| CPU | Intel notebook i5-2430M @ 2.40GHz ×2 cores |
| | Intel desktop i3-2100 @ 3.10GHz ×2 cores |
| GPU | NVIDIA notebook GeForce GT 540M @ 1.34GHz × 96 CUDA cores + i5-2430M |
| | NVIDIA desktop GeForce GTX 550Ti @ 1.8GHz × 192 CUDA cores + i3-2100 |
| | NVIDIA desktop GeForce GTX 580 @ 1.54GHz × 512 CUDA cores + i3-2100 |

Table 1 Utilized hardware devices




| Fountain image pair 3113x2111 Max disparity range: 64 | Cost Cube fill | 2 paths (horizontal) aggregation | 6 paths (2 vertical, 4 diagonal) aggregation | 8 paths (inclined diagonal) aggregation | Disparity selection |
|---|---|---|---|---|---|
| i5-2430M (Notebook CPU) | 0.939 | 1.408 | 4.592 | 13.950 | 0.112 |
| i3-2100 (Desktop CPU) | 0.723 | 1.072 | 3.408 | 11.693 | 0.087 |
| GT 540M (Notebook GPU) | 0.115 | 0.266 | 0.843 | 1.214 | 0.149 |
| GTX 550Ti (Desktop GPU) | 0.060 | 0.167 | 0.549 | 0.750 | 0.057 |
| GTX 580 (Desktop GPU) | 0.015 | 0.036 | 0.108 | 0.148 | 0.016 |

Figure 11 CPU vs GPU run time of different SGM steps

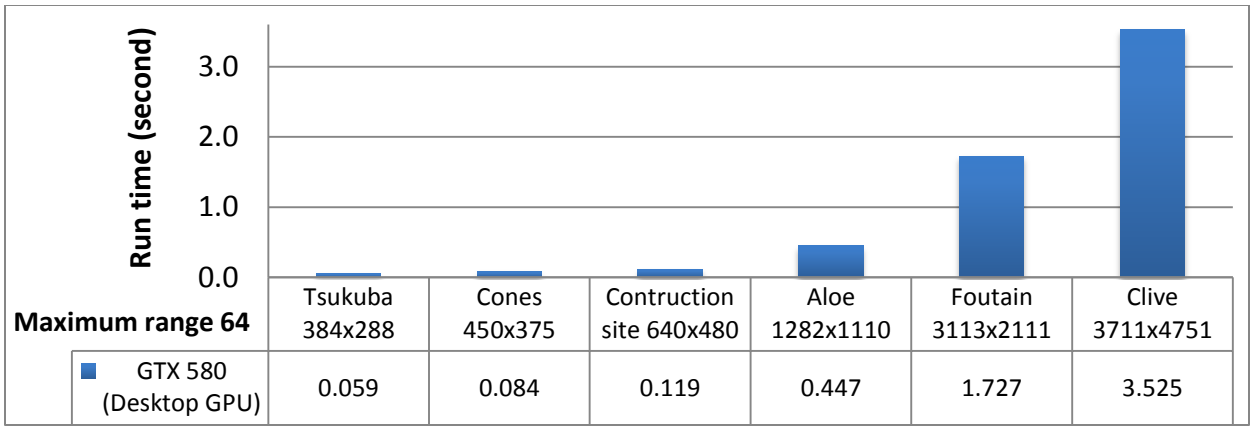



Figure 12 Total SGM run time (with 6+2 paths) of different image sizes

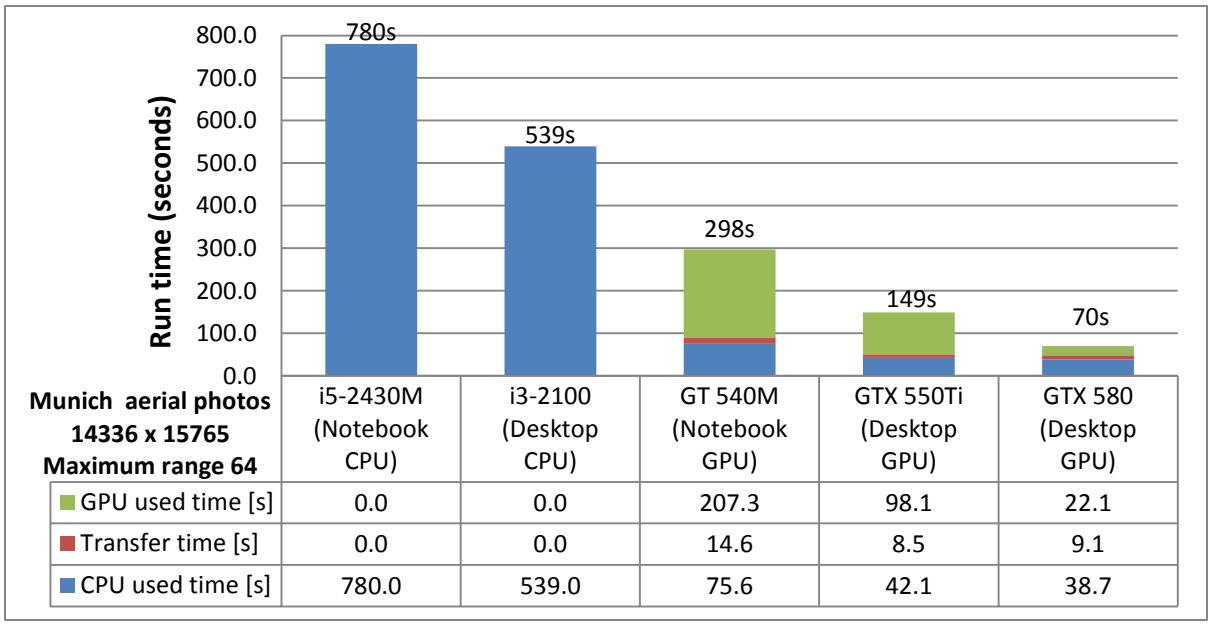



Figure 13 CPUs vs GPUs total SGM run time (with 6+2 paths) of aerial photos

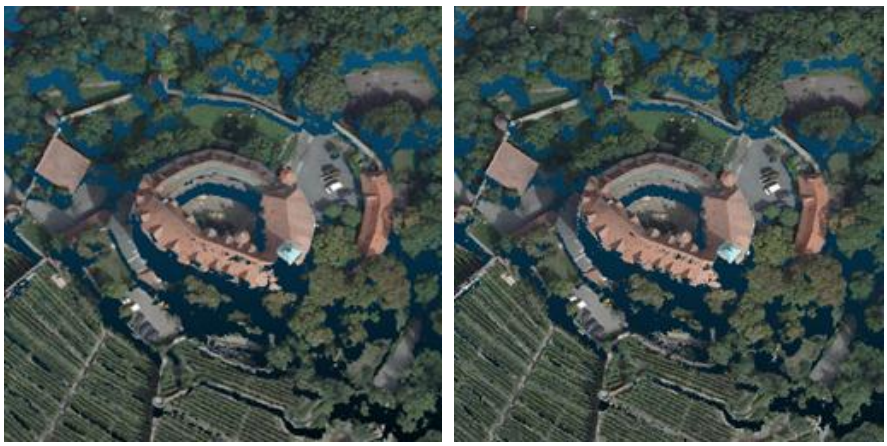


Figure 14 Vaihingen point cloud of CPU / cloud of GPU

Figure 15 Fountain point cloud of CPU / cloud of GPU

## 5. Conclusion

For speed comparison of CPU and GPU, we used an Intel i3-2100 and a NVIDIA GTX580. In pure path aggregation GTX580 shows more than 30x speed of i3-2100.

For comparison with related work, we processed small frame images. Our program is not designed for real-time application, since we want to preserve all interfaces for aerial photos which needs much GPU-CPU data transfer. Using GTX580 and a maximum search range of 64, Tsukuba 384×288 got 17 FPS, Cones 450×375 got 12 FPS, Construction site 640×480 got 8.4 FPS, and Aloe 1280×1110 got 2.2 FPS. Neglecting data transfer and speckle filtering, Tsukuba can reach 32 FPS, Cones 22 FPS, Construction site 18 FPS and Aloe 4.9 FPS. These should be enough for real-time application.

We also processed exemplary 16000×14000 size aerial images. The matching time for one pair is only 70 seconds, in which the remaining serial CPU codes have become the bottleneck.

The point clouds derived from the CPU and CUDA solutions for two data sets have shown quite similar quality, which means the CUDA results are reliable.

Overall, our GPU solution can get similar quality results as the CPU solution with much faster speed and is promising for both photogrammetric and computer vision use.

## *Bibliography*


*Harris, M. (2007).* Optimizing Parallel Reduction in CUDA. *Retrieved from Nvidia: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf*

*Hirschmueller, H. (2008). Stereo Processing by Semiglobal Matching and Mutual Information.* IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 328-341.*

*NVIDIA. (2012a).* CUDA C PROGRAMMING GUIDE. *Retrieved from Nvidia Developer Zone: http://docs.nvidia.com/cuda/cuda-c-programming-guide/*

*Rothermel, M., Wenzel, K., Fritsch, D., & Haala, N. (2012). SURE: PHOTOGRAMMETRIC SURFACE RECONSTRUCTION FROM IMAGERY.* LC3D Workshop. *Berlin.*

*Zhu, K., Butenuth, M., & d'Angelo, P. (2010). COMPUTATIONAL OPTIMIZED 3D RECONSTRUCTION SYSTEM FOR AIRBORNE IMAGE SEQUENCES.* ISPRS WG I/4: Geometric/Radiometric Modelling of Optical Spaceborne Sensors - Session III.