Technical Report
08rp015-GRIS

A Multi-View Stereo Implementation on Massively Parallel Hardware

Mate Beljan, Ronny Klowsky, Michael Goesele

September 2011
Originally written in 2008

Technische Universität Darmstadt, Germany
Department of Computer Science
Interactive Graphics Systems Group

Preliminary remark

This is a slightly updated version of a conference submission that was rejected in 2008. Since on one hand GPU technology is progressing very rapidly and on the other hand this paper is referenced on the Middlebury Multi-view Stereo benchmark page (`http://vision.middlebury.edu/mview/`) we decided to finally publish it as a technical report.

The authors, September 2011

A Multi-View Stereo Implementation on Massively Parallel Hardware

Mate Beljan, Ronny Klowsky, Michael Goesele

GRIS, TU Darmstadt

Abstract

In recent years, we have seen several approaches to implement hardware-accelerated multi-view stereo (MVS) algorithms employing the graphics processing unit (GPU) for fast and parallel computation. To our knowledge, all of them resort to various rendering passes to perform their computations. In contrast, modern GPU compute frameworks give access to the massively parallel compute capability of a GPU without forcing users to express their computations as rendering passes. This allows for a broader class of algorithms to be executed on a GPU and improves flexibility and portability.

We implemented a region-growing MVS approach for NVIDIA's CUDA framework, tested it on several data sets, and compared its computation time and reconstruction quality to an existing CPU version. For comparable reconstruction quality we often achieve a moderate speedup which is mainly limited by scarce register and memory resources of the current GPU generation.

1 Introduction

Multi-View Stereo (MVS) techniques are an active area of research (see e.g, the MVS evaluation by Seitz et al. [1, 22] for a good overview of recent results). Modern algorithms are surprisingly accurate and robust and can be applied to a wide range of input images including so called community photo collections (CPC) gathered from Internet photo sharing sites [7]. Such CPCs can consist of thousands or even millions of images so that reconstruction speed and efficiency become of prime importance.

Motivated by the drastic drop in performance increase of a single processor from 52 % per year to 20 % per year since 2002 [10], Asanovic et al. [3] argue that increasing parallelism will be the primary way to improve performance. Graphics processing units (GPUs) have traditionally been highly parallel so that the recent introduction of general purpose computing frameworks [2, 19] by the major vendors was a logical next step. Exploiting this parallelism requires, however, problems that are well suited for the GPU and implementations that match the specific constraints of the underlying hardware architecture such as memory access patterns and program flow.

We implemented the recent MVS system by Goesele et al. [7] for GPUs using NVIDIA's Compute Unified Device Architecture (CUDA) [19] and compared the performance to an optimized CPU version of the algorithm. To our knowledge, this is the first MVS implementation on GPUs that does not use the actual rendering system[1] to improve performance but relies solely on the available massively parallel compute capability. Thus the system theoretically works on other parallel systems as well and is not constrained to graphics hardware. Our contributions are as follows:

- We analyze the existing region-growing MVS algorithm [7] for parallelism and discuss different granularity options.
- We implement a GPU adapted version using the CUDA framework which processes a large number of pixels in parallel.
- We evaluate the algorithms reconstruction quality for various datasets and compare its speed to an optimized CPU implementation of the original algorithm.

The remainder of this paper is organized as follows: We first discuss related work in Section 2 before we give an overview of the basic MVS approach and a short introduction to CUDA in Section 3. We then introduce the core optimization routines of the MVS system (Section 4). Section 5 discusses implementation details and the detailed mapping of processes. Some results of our evaluation can be found in Section 6 before we conclude and discuss future work in Section 7.

[1]We do, however, use the texturing units to access bilinearly interpolated pixel values in the input images.

2 Related Work

There is a large body of previous work on multi-view stereo systems. We therefore refer to a recent survey and evaluation study by Seitz et al. [1, 22] for an overview and classification of existing techniques. We focus here on MVS approaches with GPU-based components. All of these make use of the classical rendering pipeline with programmable extensions such as vertex and fragment shaders.

The earliest MVS system on GPU by Yang et al. [26] applies a plane-sweep stereo approach. Images from multiple views are projected on planes at different candidate depths. For each pixel, the depth value with the most consistent color is selected and stored in a depth map. Subsequent approaches extend this technique by operating in a multi-resolution framework [5, 25] or by introducing multiple sweeping directions [6]. Merrell et al. [18] fuse multiple depth maps from a plane sweeping approach with a visibility-based algorithm on the GPU.

Hornung and Kobbelt [11] create an efficient pipeline for volumetric photo-consistency estimation that maps visual hull computation, visibility determination, and photo-consistency computation to various rendering passes on a GPU. Surface extraction is performed using a standard volumetric graph-cut approach [24]. Several authors propose variational multi-view stereo schemes [14, 16, 21] that again apply basic rendering techniques such as projective texture mapping or shadow mapping to accelerate the computation. Gong and Yang [8] and Zach et al. [27] describe stereo matching techniques based on dynamic programming that make extensive use of GPU computations.

Most recently, Hornung et al. [12] propose a view selection technique that selects suitable image sets for MVS reconstruction. The compute intensive parts of the algorithm are performed on the GPU.

In contrast to all of these approaches, we do not benefit from the various features of a GPU's rendering pipeline (except for bilinearly interpolated texture access). Instead, we treat it as a massively parallel processing system and implement a version of the MVS approach by Goesele et al. [7] that processes multiple pixels in parallel. The core of this approach is a per-pixel non-linear optimization of depth and surface normal. This algorithm maps well onto the CUDA compute framework [19] which is based on a single instruction multiple data (SIMD) architecture [17].

3 Overview

In this section, we briefly review the MVS approach by Goesele et al. and describe the main features of CUDA that are relevant for our implementation. Please see the original publications [7, 19] for more detailed discussions.

3.1 MVS Algorithm

The basic MVS algorithm [7] takes as input a set of images captured under controlled or uncontrolled conditions (e.g., community photo collections from Internet photo sharing sites). The images are processed with a Structure-from-Motion (SfM) system [4, 23] yielding extrinsic and intrinsic calibration data for the subset of images that were successfully registered. In addition, the matched SfM features yield a sparse representation of the scene geometry and are used to initialize the region-growing.

The core of the algorithm is the computation of depth maps for each registered input image (called a *reference view*). In the *global view selection* step, it first determines a set of neighbor views which will potentially be used in the next phase for correlation based matching. For each pixel in the reference view, the *local view selection* determines a propitious subset of neighboring views. These views are used to estimate depth and surface normal using a robust version of multi photo geometric constraint (MPGC) matching [9].

The general idea is to match pixel intensities in an $n \times n$ neighborhood around the pixel in the reference view to the corresponding patches in the neighboring views by iteratively adjusting depth, normal and color scale. The color scale models illumination differences between views. For speed and stability reasons, the normals are only updated every fifth iteration (see also the pseudo code of the algorithm in Figure 1). Pixel intensities are compared between views using normalized cross correlation (NCC) as photometric consistency measure. The process stops on convergence or when a maximum number of iterations is reached.

Instead of processing the pixels in arbitrary order, the depth map computation follows the region growing approach by [20] which processes pixels in

```
processPixel(s, t, dh, dh_s, dh_t){
  updateColorScale();
  for(int i=1; i<=MAXITERATIONS; i++){
    if(i % 5 == 0){
      updateDepthAndOrientation();
      updateColorScale();
    }
    else
      updateDepthOnly();
    computeNCC();
    checkConvergence();
  }
}
```

Figure 1: Pseudo code for depth, normal, and color scale optimization of a single pixel. Mathematical details of the functions are described in Section 4. Implementation details are given in Section 5.

the reference view in prioritized order according to their estimated matching confidence.

Merging the depth maps obtained for each input image provides a dense 3D point cloud representing the scene. With the additional knowledge of the surface normal at each point, Poisson surface reconstruction [13] can be applied to get a final mesh.

The main opportunity for parallelization of the algorithm lies in the per-pixel estimation of depth and surface normal. This requires processing a set of pixels with highest priority in parallel which may slightly decrease the quality of the results whereas offering almost linear speedup with the number of available processors.

3.2 Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) distinguishes between the *host* which runs the main program in a standard serial fashion and the massively parallel *device* (the GPU). In order to use the GPU for compute purposes, a host program can launch a *kernel* on the device which consists of a large number of threads (typically thousands or even millions of threads) executing the same program in parallel. Threads are organized in *blocks* of *warps* with currently 32 threads per warp. All threads in a warp execute the same instructions simultaneously in a SIMD fashion whereas different warps in the same block execute the instructions independently.

All threads in a block have access to a common fast *shared memory* in order to exchange data. Threads in different blocks can only exchange data via the much slower main memory of the graphics card (*global memory*). Physically, a graphics card contains multiple multiprocessors. A block is always executed on a single multiprocessor. In order to make efficient use of the hardware it is therefore essential to run at least as many blocks in a kernel as there are multiprocessors on the card. In addition, the number of threads per block should be high enough to hide latencies encountered during memory access. The number of threads per block is, however, often limited by other resources such as the number of registers or the amount of shared memory available per multiprocessor.

4 Optimization

To process a single pixel with coordinates (s, t) of the reference view (see Section 3), we optimize depth and normal of the corresponding 3D point. In order to do that an n×n neighborhood surrounding (s, t) is matched with a set of neighboring views obtained by the local view selection. With I_R and I_k denoting the intensities in the reference view and the neighboring view k, respectively, and c_k^c being the color scale, an ideal case would result in

$$I_R^c(s+i, t+j) = c_k^c \cdot I_k^c(\hat{s}+\hat{i}, \hat{t}+\hat{j}) \qquad (1)$$

for all neighboring views $k = 1 \ldots m$, for all $c \in \{0, 1, 2\}$ representing the three color channels and for all $i, j \in \{-\frac{n-1}{2} \ldots \frac{n-1}{2}\}$. The positions $(\hat{s}+\hat{i}, \hat{t}+\hat{j})$ in the corresponding neighboring views are computed using the calibration data. In the following we will omit the parameters $s, t, \hat{s}$ and $\hat{t}$ to simplify matters.

In order to find a good match, we minimize the following error derived from Equation 1 in a least squares sense for all pixels in the neighborhood

$$\sum_{k,i,j,c} (I_R^c(i,j) - c_k^c \cdot I_k^c(\hat{i},\hat{j}))^2. \qquad (2)$$

This is performed in an iterative way (see Figure 1) by either adjusting only the color scale c_k^c, by changing both depth and normal of the current sample, or by modifying only the depth.

4.1 Color Scale Update

To obtain the optimal color scale we take the derivative with respect to c_k^c, set it to zero, and solve for

c_k^c. This yields

$$c_k^c = \frac{\sum_{k,i,j,c} I_R^c(i,j) \cdot I_k^c(\hat{i},\hat{j})}{\sum_{k,i,j,c} I_k^c(\hat{i},\hat{j})^2}. \quad (3)$$

4.2 Combined Depth and Normal Update

To estimate the surface normal we encode the normal direction using per-pixel distance offsets h_s and h_t. These offsets correspond to the per-pixel rate of change of depth in the s and t directions, respectively (see Figure 2). Linearizing Equation 1 with respect to the depth leads to

$$\begin{aligned} I_R^c(i,j) = {} & c_k^c \cdot I_k^c(\hat{i},\hat{j}) + \\ & c_k^c \cdot \frac{\partial I_k^c(\hat{i},\hat{j})}{\partial h} \cdot (dh + i\, dh_s + j\, dh_t) \end{aligned} \quad (4)$$

where dh, dh_s, dh_t are the differences of the optimal values to the initial estimations of h, h_s, h_t, respectively.

Equation 4 can be regarded as a system of $3mn^2$ linear equations for the three unknowns dh, dh_s, dh_t. With

$$A = c_k^c \begin{pmatrix} \frac{\partial I_k^c(\hat{i},\hat{j})}{\partial h} & i\frac{\partial I_k^c(\hat{i},\hat{j})}{\partial h} & j\frac{\partial I_k^c(\hat{i},\hat{j})}{\partial h} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix}$$

$$b = \begin{pmatrix} I_R(i,j) - I_k^c(\hat{i},\hat{j}) \cdot c_k^c \\ \cdots \\ \cdots \end{pmatrix}$$

$$x = (dh, dh_s, dh_t)^{\mathrm{T}}$$

we can rewrite Equation 4 as

$$Ax = b. \quad (5)$$

From Madsen et al. [15] we know that the optimal solution x^* in a least squares sense solves

$$(A^{\mathrm{T}}A)x^* = A^{\mathrm{T}}b. \quad (6)$$

The matrix $A^{\mathrm{T}}A$ is a symmetrical 3×3 matrix and can thus be easily inverted if $\det(A^{\mathrm{T}}A) \neq 0$. Matrix A can be decomposed in matrices A_k containing only the rows of the kth view. This yields $A^{\mathrm{T}}A$ as the componentwise sum

$$A^{\mathrm{T}}A = \sum_{k=1}^{m} A_k^{\mathrm{T}} A_k. \quad (7)$$

We will use this decomposition to compute $A^{\mathrm{T}}A$ efficiently in parallel.

4.3 Depth Only Update

In the majority of iterations, we keep the normal fixed, i.e., the estimates for h_s and h_t, and update only the depth. Linearizing Equation 1 we thus only introduce dh while the terms dh_s and dh_t in Equation 4 are dropped. We therefore minimize

$$\sum_{k,i,j,c} (I_R^c(i,j) - c_k^c \cdot I_k^c(\hat{i},\hat{j}) - c_k^c \cdot \frac{\partial I_k^c(\hat{i},\hat{j})}{\partial h} \cdot dh)^2. \quad (8)$$

Similarly to updating the color scale (Section 4.1) we take the derivative with respect to the depth h, set it to zero, and solve for dh resulting in

$$dh = \frac{1}{\sum_{k,i,j,c} \left(c_k^c \frac{\partial I_k^c(\hat{i},\hat{j})}{\partial h}\right)^2} \cdot \sum_{k,i,j,c} (I_R^c(i,j) - c_k^c \cdot I_k^c(\hat{i},\hat{j}))\, c_k^c \frac{\partial I_k^c(\hat{i},\hat{j})}{\partial h}. \quad (9)$$

4.4 Computing NCC

Computation of the convergence criterion and the photometric consistency measure relies on the normalized cross correlation between the neighborhood in the reference view and an individual neighboring view k. The NCC between two patches containing pixel intensities $\{x_i\}_{i=1\ldots n}$ and $\{y_i\}_{i=1\ldots n}$ is defined as

$$NCC_k = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \cdot \sum_{i=1}^{n}(y_i - \bar{y})^2}} \quad (10)$$

where $\bar{x}$ and $\bar{y}$ are the mean intensity values in each patch.

5 Massively Parallel Implementation

As discussed in Section 3.1, the goal of our implementation is to process multiple pixels in parallel. This can be done at different granularity, e.g.,

- one thread per pixel,
- one thread per pixel and neighboring view, or
- one thread per pixel, neighboring view, and element in the n×n neighborhood of the pixel.

Efficient processing using only one thread per pixel requires processing a large number of pixels in parallel. Each thread then uses a large number of scarce resources (shared memory, registers) which limits

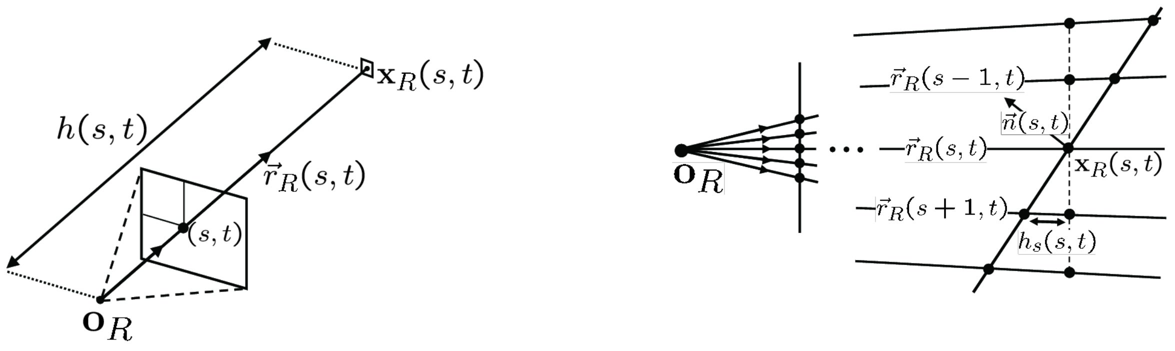

Figure 2: Encoding surface normal using h_s and h_t [7]. *Left:* The window centered at pixel (s,t) in the reference view corresponds to a point $\mathbf{x}_R(s,t)$ at a distance $h(s,t)$ along the viewing ray $\vec{r}_R(s,t)$. *Right:* Cross-section through the window to show parametrization of the window orientation as depth offset $h_s(s,t)$.

the number of threads actually running in parallel. Therefore, performance is likely to be bad.

On the other hand, using one thread per pixel, neighboring view, and element is also expected to have sub-optimal performance. The limitation in this case is the remaining serial fraction – the part of the algorithm that can only be executed by a single thread per pixel (see Chapter 2.5 of Mattson et al. [17] for a discussion of this effect).

We therefore choose the intermediate solution with one thread per pixel and neighboring view. The optimization is performed by a single kernel which receives as arguments an array of positions and initial values for h, h_s, and h_t as well as the number of pixels in the array that should be processed. Upon completion it returns the computed parameters and confidence values which are processed by the host program. This includes copying of results to the output depth map and adding neighbors to the priority queue as in the original algorithm [7].

5.1 Memory Layout

We store and access all images on the GPU as textures using either nearest neighbor sampling (reference view) or bilinear interpolation (neighboring views). The corresponding projection matrices are stored in *constant memory*. Since we need to access the same image values and image gradient values for the n×n neighborhood in all neighboring views multiple times per iteration, we keep a cached version in global memory. This cached version is updated once per iteration saving a large amount of computations (mainly projection of the point coordinates into the neighboring views) and texture accesses. Note that access to global memory and texture access have the same latency if the sample is not in the texture cache so that this saves mostly computation time plus some accesses to constant memory to retrieve the projection matrices. All other per-thread or per-pixel values are stored in *registers* or *shared memory*.

5.2 Parameter Update

Figure 3 shows the pseudo code for the function call that updates dh, dh_s, and dh_t according to Section 4.2. This code is executed by each thread, i.e., once for each pixel and neighboring view. It first computes the elements of ${A_k}^{\mathrm{T}}A_k$ and ${A_k}^{\mathrm{T}}b$ independently for each thread. Note that ${A_k}^{\mathrm{T}}A_k$ is symmetric so that only elements of the upper diagonal are computed. In the next step, the contributions of the neighbors to $A^{\mathrm{T}}A$ and $A^{\mathrm{T}}b$ are summed using multiple *reductions* (a tree-based reduction can compute the sum of 2^n numbers in n steps using $n/2$ threads, see Mattson et al. [17] for details). The 3×3 matrix $A^{\mathrm{T}}A$ is then inverted explicitely (in the case of $\det(A^{\mathrm{T}}A) \neq 0$) and the updates of dh, dh_s, and dh_t are computed serially using a single thread per pixel.

Other functions that compute only the depth update dh, the color scale c_k, or the normalized cross correlation and confidence values operate according to the same pattern. They first compute the contribution of each neighboring view k in parallel and collect the results per pixel using one or multiple reductions.

```
updateDepthAndOrientation(int windowNrBlock, int neighborNr) {
  // compute matrix and vector elements, note that $A^T A$ is symmetric
  float3 a11=a12=a13=a22=a23=a33=a1b=a2b=a3b=0;
  for (int index=0; index < WINDOWSAMPLES; index++) {
    // read image data for reference view and current neighbor
    float3 IR = I_R(windowNrBlock, index);
    float3 Ik = I_k(windowNrBlock, neighborNr, index);
    float3 Ik_dh = I_k_dh(windowNrBlock, neighborNr, index)
                              * c_k[windowNrBlock][neighborNr];

    a11 += Ik_dh*Ik_dh;
    a12 += Ik_dh*(Ik_dh*offset_i(index));
    a13 += Ik_dh*(Ik_dh*offset_j(index));
    a22 += (Ik_dh*offset_i(index)) * (Ik_dh*offset_i(index));
    a23 += (Ik_dh*offset_i(index)) * (Ik_dh*offset_j(index));
    a33 += (Ik_dh*offset_j(index)) * (Ik_dh*offset_j(index));

    a1b += Ik_dh*(IR - Ik*c_k[windowNrBlock][neighborNr]);
    a2b += (Ik_dh*offset_i(index)) * (IR - Ik*c_k[windowNrBlock][neighborNr]);
    a3b += (Ik_dh*offset_j(index)) * (IR - Ik*c_k[windowNrBlock][neighborNr]);
  }

  // sum contribution from all neighbors using reduction; store in ata, atb
  shared float3 sum[windowNrBlock][neighborNr]=a11;
  for(unsigned int s = NEIGHBORCOUNT/2; s > 0 ; s >>= 1) {
    if(neighborNr < s) {
      sum[windowNrBlock][neighborNr] += sum[windowNrBlock][neighborNr + s];
    }
  }
  ata[windowNrBlock][0] = sum[windowNrBlock][0];
  // same reduction for the remaining elements of ata, atb
  ...

  // invert the symmetric matrix $A^T A$ explicitly if det($A^T A$)!= 0 and
  // solve for the depth and orientation parameters using a single thread
  if(neighborNr == 0) {
    ...
  }
}
```

Figure 3: Pseudo code for *updateDepthAndOrientation*. Each thread executes this function for its pixel position (encoded as *windowNrBlock*) and neighboring view (encoded as *neighborNr*). *WINDOWSAMPLES* is the number of elements in the neighborhood around the pixel, i.e., *WINDOWSAMPLES*=n×n=n^2. *NEIGHBORCOUNT* is the number of neighboring views currently active.

5.3 Local View Selection

Local view selection plays an important role in the original MVS algorithm [7] since it allows to exclude non-matching views from the optimization. It also reduces the computational cost by operating only on a subset of neighboring views (the *active views*) and ensures parallax between these views.

We implemented a modified version of local view selection. Initially, we use all neighboring views for each pixel's optimization and remove views form the active set if the photometric consistency measure fails for that view. The optimization stops if less than a given number of views are active. This implementation avoids complex local view selection operations on the GPU. Since samples are computed from more neighboring views, the quality of the depth maps typically improves.

Unless noted otherwise, we used four active views out of eight neighboring views selected by the local view selection for CPU code. The GPU works as above requiring at least four active views.

6 Results

We compare our work with the results of an improved implementation of the MVS code provided by the authors [7]. Due to various optimizations, this code is about ten times faster than the original code while maintaining the quality of the results. In addition, radial distortion of input images from CPCs is no longer corrected in a preprocessing step relying on a database of known camera parameters. Instead, the radial distortion parameters are included in the estimation of intrinsic parameters during the SfM phase which improves overall accuracy.

All results were computed on a Linux PC with an Intel Quadcore CPU with 2.66 GHz equipped with a NVIDIA GeForce 8800 GT and a NVIDIA Tesla C870. The Tesla C870 was used to run the CUDA Version 1.1 code since it is only used for computing and does not run the graphical display. It contains 16 multiprocessors with 8192 registers and 16384 bytes of shared memory per multiprocessor.

We tested the system on three datasets: The templeFull dataset from the Middlebury benchmark [1], the Florence dataset consisting of 458 images of the Duomo in Florence captured under outdoor conditions, and the CPC of the Statue of Liberty downloaded from Internet photo sharing sites.

6.1 Registers vs. Local Memory

The number of pixels processed in parallel on a single multiprocessor is limited by the number of available registers or the amount of shared memory. CUDA allows to limit the number of registers per thread at compile time using the *maxrregcount* compiler flag.

We tested several configurations with different number of registers per thread (see Table 1). In each case, we selected the maximum number of pixels per thread while avoiding partially filled warps, i.e., the number of pixels must be a multiple of four. Given constant memory access time, all configurations would have almost identical run time.

The 16×8 configuration with 62 registers per thread maximizes the amount of registers while minimizing local memory usage and yields the best performance. We therefore performed all further reconstructions with this configuration. Note that the 32×8 configuration returns an invalid number of reconstructed samples due to an error in CUDA.

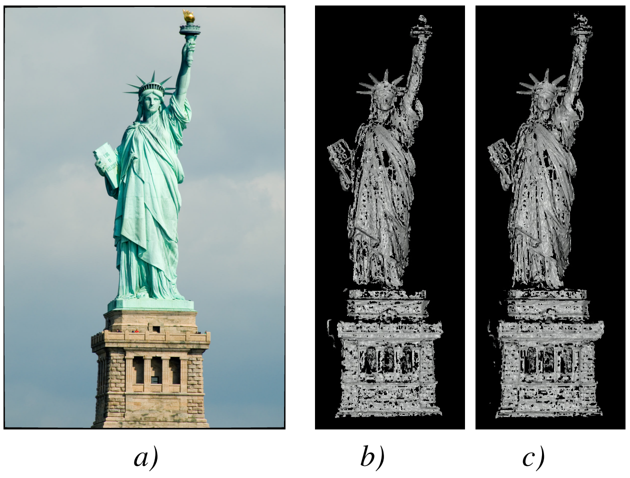

Figure 4: View from the Statue of Liberty dataset and reconstructed depth maps. *b)* Optimized CPU version with 8 neighbors and view selection (113261 samples). *c)* GPU version with 8 neighbors (119603 samples).

6.2 Performance on CPU vs. GPU

Tables 2 and 3 show the performance of various versions of the algorithm. There are two versions – original and optimized – of the CPU implementation with active local view selection. Additionally, we modified the optimized CPU implementation to work with the local view selection algorithm of the GPU version as described in Section 5.3. If this algorithm is initialized with only four neighboring views, local view selection is effectively turned off.

The original CPU implementation is about an order of magnitude slower than all other versions. Regarding time per reconstructed sample, our GPU version is faster than the comparable CPU version except for the Florence dataset with eight neighbors. This exception is probably caused by the low number of actually reconstructed samples (about 10% less than the other versions).

Applying local view selection in the CPU case results in an additional speedup as already discussed in the original paper by Goesele et al. [7].

6.3 Quality of Reconstruction

As shown in Table 2, the number of reconstructed samples in the templeFull dataset is almost equal for the corresponding CPU and GPU versions. This is also visually apparent in Figure 5 which shows the reference view and renderings of the reconstructed depth maps.

configuration (pixels×neighbors)	**registers per thread**	**registers per block**	**local memory per thread**	**shared memory per block**	**total GPU time**	**recon-structed samples**
8×8	83	5312	120 byte	2876 byte	0.592037 s	2191
16×8	62	7936	120 byte	5700 byte	0.496031 s	2191
16×8	48	6144	200 byte	5700 byte	0.516032 s	2191
32×8	32	8192	716 byte	11348 byte	0.464029 s	1843
44×8	16	5632	760 byte	15584 byte	0.864054 s	2191

Table 1: Run times for processing the initial SfM features of View 123 from the templeFull dataset [1] using different configurations of threads. We varied the number of registers per thread using the compile flag *maxrregcount* which shifts register content to local memory. The number of threads per block is always a multiple of the warp size (32) to avoid partially filled warps. Note that the configuration 32×8 uses all available registers but yields an invalid number of reconstructed samples due to an error in CUDA. The last configuration is limited to 44×8 threads due to insufficient shared memory.

templeFull dataset	**execution time**	**GPU time**	**reconstructed samples**	**time per sample**
optimized CPU with 4 neighbors	23.26 s	—	83673	0.278 ms
GPU with 4 neighbors (32×4)	20.81 s	12.36 s	80972	0.257 ms
original CPU with view selection	335.38 s	—	92823	3.613 ms
optimized CPU with view selection	30.27 s	—	99511	0.304 ms
optimized CPU with 8 neighbors	53.40 s	—	99371	0.537 ms
GPU with 8 neighbors (16×8)	49.06 s	34.53 s	99623	0.492 ms

Table 2: Run times and number of reconstructed samples for the MVS algorithms operating on View 123 of the templeFull dataset [1]. *Execution time* gives the overall process time of the MVS algorithm without file access. *GPU time* only measures the kernel time and memory transfer between host and GPU. *Reconstructed samples* gives the number of samples in the computed depth map. *Time per sample* is the ratio of execution time to number of reconstructed samples. The different algorithms are described in Section 6.2.

Florence dataset	**execution time**	**GPU time**	**reconstructed samples**	**time per sample**
optimized CPU with 4 neighbors	90.39 s	—	318916	0.283 ms
GPU with 4 neighbors (32×4)	72.42 s	42.53 s	281998	0.257 ms
optimized CPU with view selection	201.45 s	—	616780	0.327 ms
optimized CPU with 8 neighbors	284.39 s	—	618951	0.459 ms
GPU with 8 neighbors (16×8)	283.17 s	194.03 s	563047	0.503 ms

Table 3: Run times and number of reconstructed samples for the MVS algorithms operating on a view from the Florence dataset. See Table 2 for a description of the different columns.

In contrast to that, there is a clear difference in the number of reconstructed samples for the other two examples. While the view from the Florence dataset (see Table 3 and Figure 6) has less reconstructed samples in the GPU version, the reconstructed view from the Statue of Liberty dataset (Figure 4) has more samples in the GPU version.

For an objective quality measure, we submitted a version of the reconstructed templeFull dataset to the Middlebury multi-view stereo evaluation site [1]. We achieved an accuracy of 0.79mm and a completeness of 92.2%.

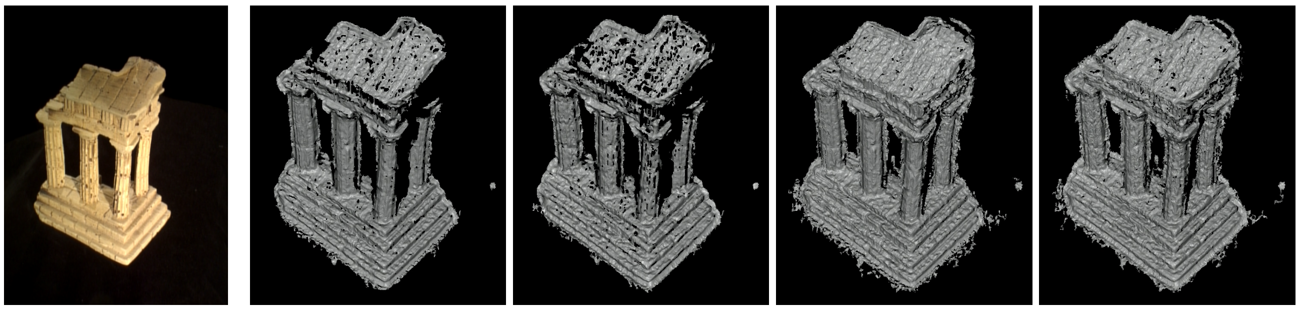

Figure 5: View 123 from the templeFull dataset [1] and reconstructed depth maps. *b)* Optimized CPU version with 4 neighbors. *c)* GPU version with 4 neighbors. *d)* Optimized CPU version with 8 neighbors and view selection. *e)* GPU version with 8 neighbors.

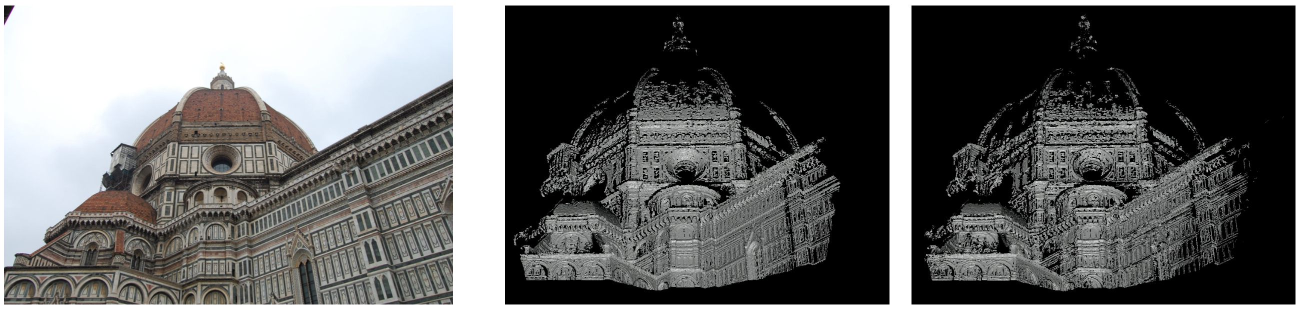

Figure 6: View from the Florence dataset and reconstructed depth maps. *b)* Optimized CPU version with 8 neighbors and view selection. *c)* GPU version with 8 neighbors.

7 Conclusion and Future Work

We showed that it is possible to implement a region-growing multi-view stereo system on a GPU using the CUDA framework. Unlike previous hardware accelerated multi-view stereo algorithms, this problem does not naturally and efficiently map to multiple rendering passes due to the required processing order defined by the priority queue. Note that grave violation of processing order will often result in reconstruction faults caused by convergence of the non-linear optimization to wrong local minima. Our algorithm processes only a small number of pixels from the top of the priority queue in parallel thereby approximately fulfilling the ordering constraint. It therefore yields different but still high-quality reconstruction results compared to a serial version.

As shown in Section 6, we achieve comparable computation times to the strongly optimized CPU version. In various cases, we even surpass the performance of the optimized CPU version by up to 10 %. The GPU performance of the studied MVS algorithm mostly depends on memory access latencies. Since the number of registers and the amount of shared memory is insufficient on current hardware, we are forced to rely on slow global or local memory in the computations. Due to the same reasons, we are unable to compensate for the resulting latency by running more threads in parallel.

Since scarce resources on a multiprocessor are a common bottleneck in CUDA implementations, we expect that future hardware generations will alleviate this problem. The proposed algorithm will immediately benefit from this development without redesign. In contrast, reaching a similar speedup for the already optimized CPU version will require substantial changes and efforts as predicted by Asanovic et al. [3].

References

[1] Multi-view stereo evaluation web page. http://vision.middlebury.edu/mview/.

[2] AMD. AMD stream computing: Software stack. White paper, AMD, 2007.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[4] M. Brown and D. G. Lowe. Unsupervised 3D object recognition and reconstruction in unordered datasets. In *Proc. 3DIM*, pages 56–63, 2005.

[5] N. Cornelis and L. V. Gool. Real-time connectivity constrained depth map computation using programmable graphics hardware. In *Proc. CVPR*, pages 1099–1104, 2005.

[6] D. Gallup, J.-M. Frahm, P. Mordohai, Q. Yang, and M. Pollefeys. Real-time plane-sweeping stereo with multiple sweeping directions. In *Proc. CVPR*, 2007.

[7] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. M. Seitz. Multi-view stereo for community photo collections. In *Proc. ICCV*, 2007.

[8] M. Gong and Y.-H. Yang. Near real-time reliable stereo matching using programmable graphics hardware. In *Proc. CVPR*, pages 924–931, 2005.

[9] A. Gruen and E. Baltsavias. Geometrically constrained multiphoto matching. *Photogrammetric Engineering and Remote Sensing*, 54(5):633–641, May 1988.

[10] J. Hennessey and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 2007.

[11] A. Hornung and L. Kobbelt. Robust and efficient photo-consistency estimation for volumetric 3d reconstruction. In *Proc. ECCV*, pages 179–190, 2006.

[12] A. Hornung, B. Zeng, and L. Kobbelt. Image selection for improved multi-view stereo. In *Proc. CVPR*, 2008.

[13] M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson surface reconstruction. In *Proc. SGP*, pages 61–70, 2006.

[14] P. Labatut, R. Keriven, and J.-P. Pons. Fast level set multi-view stereo on graphics hardware. In *Proc. 3DPVT*, pages 774–781, 2006.

[15] K. Madsen, H. B. Nielsen, and O. Tingleff. Methods for non-linear least squares problems (2nd ed.). Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2004.

[16] J. Mairal, R. Keriven, and A. Chariot. Fast and efficient dense variational stereo on gpu. In *Proc. 3DPVT*, pages 97–104, 2006.

[17] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, 2004.

[18] P. Merrell, A. Akbarzadeh, L. Wang, P. Mordohai, and J.-M. Frahm. Real-time visibility-based fusion of depth maps. In *Proc. ICCV*, 2007.

[19] NVIDIA. *CUDA Programming Guide Version 1.1*, 2007.

[20] G. P. Otto and T. K. W. Chau. 'Region-growing' algorithm for matching of terrain images. *Image Vision Comput.*, 7(2):83–94, 1989.

[21] J.-P. Pons, R. Keriven, and O. Faugeras. Modelling dynamic scenes by registering multi-view image sequences. In *Proc. CVPR*, pages 822–827, 2005.

[22] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Proc. CVPR*, pages 519–528, 2006.

[23] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections in 3D. In *Proc. SIGGRAPH*, pages 835–846, 2006.

[24] G. Vogiatzis, P. H. S. Torr, and R. Cipolla. Multi-view stereo via volumetric graph-cuts. In *Proc. CVPR*, pages 391–398, 2005.

[25] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *Proc. CVPR*, pages 211–220, 2003.

[26] R. Yang, G. Welch, and G. Bishop. Real-time consensus-based scene reconstruction using commodity graphics hardware. *Computer Graphics Forum*, 22(2):207–216, 2003.

[27] C. Zach, M. Sormann, and K. Karner. Scanline optimization for stereo on graphics hardware. In *Proc. 3DPVT*, pages 512–518, 2006.