Photogrammetric Week '15 Dieter Fritsch (Ed.) Wichmann/VDE Verlag, Belin & Offenbach, 2015

Real-time Photometric Stereo

Andrew Pevar, Lieven Verswyvel, Stamatios Georgoulis,
Nico Cornelis, Marc Proesmans, Luc Van Gool, Leuven

ABSTRACT

Dome-shaped devices consisting of a single digital camera and multiple light sources have been used in the past for the 3D scanning of objects. They leverage Photometric Stereo techniques in order to build detailed 3D models of these objects. Their advantage is that they can pick up even subtle details of the shape. Yet, these systems typically suffer from high recording and processing times. This paper introduces a novel GPU-accelerated implementation that calculates the shape normals, as well as the albedo and ambient lighting through the Photometric Stereo technique, providing to users the ability for real-time feedback on the recording process. An originally serial algorithm was mapped to the architecture of an NVIDIA GPU and the CUDA programming platform. To maximize performance, various optimizations were applied, like reducing the total amount of memory accesses, coalescing the memory accesses into the minimal number of transactions, reducing register usage to avoid spilling, hiding latency and maximizing thread occupancy. Our method reduces the processing time, accelerating the original implementation by a factor of 950, thereby altering the way in which such devices can be used.

1. INTRODUCTION

Photometric Stereo methods recover the 3D shape and albedo of an object using multiple images in which typically the viewpoint is fixed and only the lighting conditions vary (Basri et al., 2006). The technique is based on the fact that the amount of light reflected by a surface depends on the orientation of the surface in relation to the camera and the light source. Like others, the dome-shaped device we use, consists of a single camera on top of a hemisphere with LED light sources inside. With this setup, the position of the object and the camera can be kept constant, while varying the position and angle of the light source by subsequently activating the different LEDs.

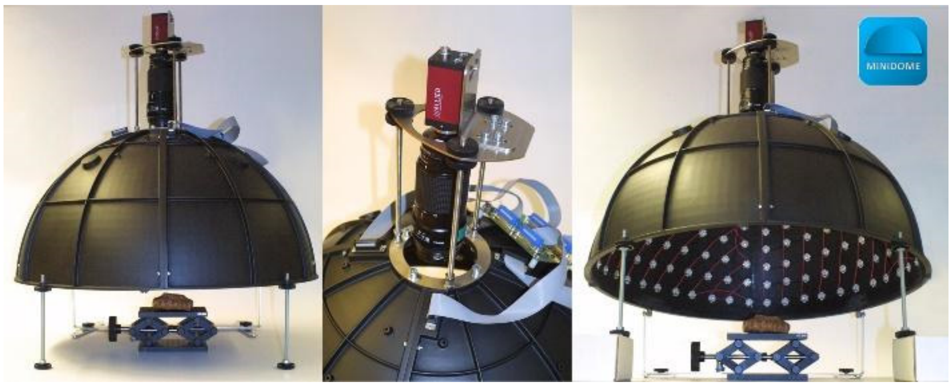

Figure 1: The Minidome (VISICS, 2015). *Left:* the camera on top looks vertically down upon the object, positioned close to the center of the hemisphere. *Middle:* detailed view of the camera, equipped with a good quality lens. *Right:* inside view of the dome, with regularly placed LEDs.

Such devices are currently used for digitizing a wide range of objects that are difficult to capture accurately using conventional techniques. They have a broad range of applications in different fields. In archeology, the recovered results allow for photo-realistic virtual renderings of the scanned objects in 2.5D, offering an interactive way to virtually inspect these objects. This makes them ideal for studying and demonstrating fragile historic artefacts such as clay tablets, cachets, manuscripts and coins. In biology, they are used for digitizing insects, which are not only fragile, but usually also have complex microstructures on the surface of their wings, making their digitization more difficult. In manufacturing, they are used to inspect the surface finish of industrial parts (for example in automotive applications), etc.

Up until now, the time to process the input images and perform the Photometric Stereo technique ranged from 5 minutes up to 15 minutes, depending on the resolution of the recorded images. Therefore, the processing of the data is usually scheduled to run as a batch job at night. This approach is time-consuming and does not allow for any timely quality control or feedback from the user.

In this paper we present a novel method to perform Photometric Stereo efficiently on a GPU. Our starting point is an existing CPU-only implementation (Willems et al., 2005). The proposed algorithm is designed to process the data generated by dome-shaped devices like the one already described, i.e. KU Leuven's `Minidome' (Willems et al., 2005 and Watteeuw et al., 2013). The goal of our work is to produce detailed 3D models, while drastically reducing the time needed to produce these results (i.e. real-time).

In this work two approaches are discussed, both solving a different need. The first approach focuses on processing the entire image set as quickly as possible. The second approach produces intermediate results as more data is being fed into the system. This allows for instant feedback on the recording process, enabling the operator to abort prematurely, e.g. to make adjustments to the camera settings or if the intermediate results are deemed satisfactory already. By providing immediate feedback, the recording can be calibrated more quickly, increasing overall productivity. This makes such devices a more attractive option for both researchers and consumers.

In the following sections we will analyze how the system is designed and implemented, and which optimizations were applied to achieve the necessary speedup. A visualization tool is described to provide the user with immediate feedback and fully utilize the real-time nature of the solution. Finally, we present the results that were obtained using our method.

2. RELATED WORK

In recent years, a large body of work has explored how to use GPUs for general-purpose computing, also known as GPGPU. Before the advent of general-purpose languages for GPGPU, GPU implementations could only be achieved using existing 3D-rendering APIs such as OpenGL or DirectX. Recognizing the value of GPUs for general-purpose computing, GPU vendors added driver and hardware support to use the highly parallel hardware of the GPU without the need for computation to proceed through the entire graphics pipeline or the need to use 3D APIs at all. A wide variety of applications have achieved drastic speedups with GPGPU implementations with the most popular being Neural Networks. Unsurprisingly, Image Processing has benefited from this trend too. Attila Remenyi used GPGPU to process biomedical image data (Remenyi, 2011), Hiren Patel accelerated the processing of polarimetric images for defense-related purposes (Patel, 2010) and Yanqing et al. utilized GPGPU for the processing of palmprint images (Yanqing et al., 2012), to just give a couple of application-ready examples.

Due to the highly increased performance with the use of GPGPU, a number of studies were conducted to quantify what the possible gains would be. Che *et al.* did such a performance study on GPGPU applications, comparing the performance of CPU and GPU implementations of six naturally data-parallel applications (Che et al., 2008). The architecture of GPUs differs greatly from general-purpose CPUs, so optimized implementations differ for both as well. Some general optimization principles have been researched by Ryoo *et al.* (Ryoo et al., 2008). Other key factors in pushing GPGPU performance, such as memory access optimization, have been researched by Bibikov et al. (Bibikov et al., 2011). The benefits of using multiple GPU's for Image Processing were researched by Song & Biao (Song & Biao, 2012).

The work presented in this paper is based on the Photometric Stereo implementation described in the work of Willems et al. (Willems et al., 2005) and follow up work (Watteeuw et al. 2014). To the best of our knowledge, there are only a few works that have tried to speed-up the Photometric Stereo process. Malzbender et al. (Malzbender et al., 2006) used a high-speed video camera, computer controlled light sources and GPU implementations of the algorithms to accelerate Photometric Stereo methods. Their approach only works for low-resolution images. Schindler introduced a method which uses the light emitted by a computer screen to illuminate an object such as a human face from multiple directions, simultaneously capturing images with a webcam in order to perform Photometric Stereo (Schindler, 2008). This CPU implementation was still rather slow, but Nozick (Nozick, 2010) managed – with a similar setup – to render human faces in real-time, thanks to a pyramidal integration of the normal maps based on an iterative scheme. An inherent limitation of this approach is that it is mainly tied to human face recordings. Finally, Varnavas et al. have implemented a GPU version of Photometric Stereo (Varnavas et al., 2010). However, their solution was designed for a different scanning setup that only captures a maximum of 128 images at a 0.3 MP resolution.
In general, compared to previous approaches our method offers a scalable solution towards not only more images but also images larger in resolution. Our novel implementation of the Photometric Stereo scheme is only bound by transfer speeds over the PCI bus. The performance boost using our approach allows for the first time to adapt and alter the recording process based on the feedback of the user.

3. SYSTEM OVERVIEW

3.1. Output of the algorithm

The implemented algorithm produces three output images: a normal image, an albedo image and an ambient image (Willems et al., 2005 and Watteeuw et al., 2013). The normal image is a two-valued representation of the geometric details of the digitized object. It contains the surface orientation for each pixel and thus the shape of the object.

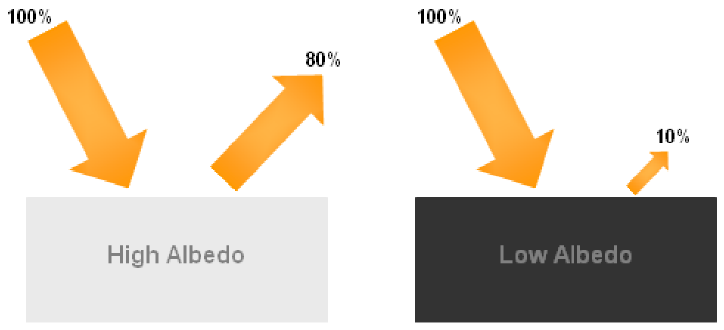

Figure 2: Illustration of albedo (NC State University, 2015).

The albedo image contains the diffuse reflection coefficient for each pixel, providing information about the optical characteristics of the object's surface. The amount of energy that is reflected by a surface is determined by the reflectivity of that surface, called the albedo. A high albedo means the surface reflects the majority of the radiation that hits it and absorbs the rest (NC State University, 2015). An example illustrating this principle is shown in Figure 2.
The ambient image mimics the object's observed color when illuminated under ambient lighting conditions, in this particular case it simulates the effect of all dome LEDs lighting the object simultaneously. The albedo and normal map are used to create an accurate textured 3D model of the

digitized object. The ambient map serves as a comparison with / and alternative for albedo, for inspection purposes for our users. Examples of the three result images are pictured in Figure 3.

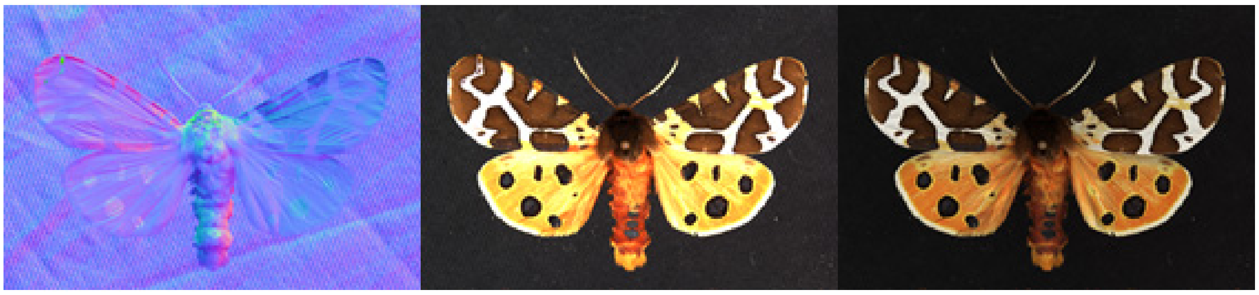

Figure 3: The normal (left), albedo (center) and ambient (right) result images for the scan of a moth.

3.2. Overview of the algorithm

The photometric stereo algorithm operates on a set of 260 images from the camera – each with one LED activated - to produce three output images, respectively containing the normal, the albedo and the ambient information for the observed object. In a real-time context, also the bayer pattern of the camera image needs to be taken into account. The algorithm has a number of internal dependencies, which allow for a modularized approach. These dependencies are visualized in Figure 4.

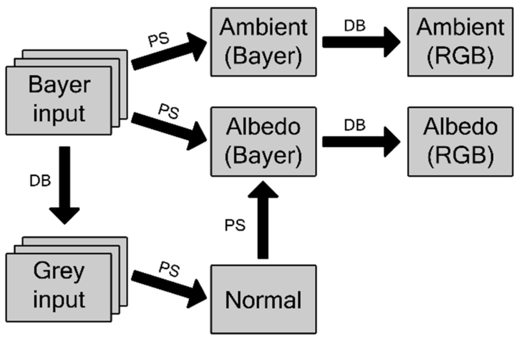

Figure 4: Dependencies in Photometric Stereo. PS = Photometric Stereo, DB = Debayering.

In terms of computational complexity, the ambient output is the most straightforward to be processed. In order to compute the valid normal for each pixel, the input image set must be demosaiced to an image of gray-values in a pre-processing step. Similarly, Photometric Stereo produces bayer images for the albedo and ambient results. These require a post-processing step to produce the desired color images. Image processing libraries, such as OpenCV, provide methods for performing this demosaicing step. Contrary to the algorithm itself, the demosaicing is pixel inter-dependent. For optimal efficiency, these steps should be completed separately. The albedo of a pixel, calculated using the geometry of the observed object, can only be computed when its corresponding normal has been found.

The observed intensity for a given pixel using the Lambertian lighting model is related to the angle between the incoming light and the normal at that position.

$$\bar{n}^T(x) * \bar{L}_p(x) * \bar{a}(x) = \bar{I}_p(x) \quad (1)$$

Where a(x) is the albedo at point x, $\bar{L}_p(x)$ is the light vector observed at pixel x when illuminated with a LED at light position p, $\bar{I_p}(x)$ are the observed image intensities (by default RGB values), under the same lighting considerations. These color intensities $\bar{I_p}(x)$ are converted to the corresponding gray intensity $G_p(x)$ using the standardized Rec. 601 luma equation (Poynton, C., 2012), for speed and memory considerations.
Next, since the equation holds both an unknown normal as well as an albedo, we solve for $\bar{n}'^T_i = a(x) * \bar{n}^T_i(x)$, holding both unknowns.

$$\bar{n}'^T(x) * \bar{L}_p(x) = G_p(x) \quad (2)$$

The set of all light positions P each contribute an equation to the system that has to be solved for $n'(x)$. The iterative approach outlined below shows how better approximations for $n'(x)$ can be achieved by removing equations from the system when they are deemed to be outliers and do not fit the model well. Equations are removed if the observed gray value is under- or over-saturated (based on chosen thresholds t_{Min} and t_{Max}) or if the gray value lies too far from the predicted gray value $G^*_p(x)$ (which is determined by the empirical residual threshold t_{res}). The set $P_L \subseteq P$ represents all light positions that have not yet been removed and for which equation (1) holds true. After a number of iterations the normal converges and P, the set of light positions that produce valid equations, remains the same.
The iterative process to solve for equation (2) is as follows:

```
$P_L \leftarrow \{p \in P \mid t_{\mathrm{Min}} < G_p(x) < t_{Max}\}$

$for\ i \leftarrow 1\ to\ 10:$
    $minimize\ \sum_{p \in P_L} \left(\bar{n}'^T_i(x) * \bar{L}_p(x) - G_p(x)\right)^2\ and\ solve\ for\ \bar{n}_i'(x)$

    $\forall\, p \in P_L$:
        $if\ \left(\bar{n}'^T_i(x) * \bar{L}_p(x) - G_p(x)\right)^2 < t_{res}$
        $then\ P_L \leftarrow P_L \setminus \{p\}$
```

The normal can be extracted by simple normalization:

$$\bar{n}^T_i(x) = \frac{\bar{n}'_i(x)}{||\bar{n}'_i(x)||}$$

Given the normal, the albedo $a_{rgb}(x)$ can be estimated from equation (1).

$$\text{minimize } \left(\bar{n}^T(x) * \bar{L}_p(x) * \mathrm{a_{rgb}(x)} - I_{p,rgb}(x)\right)^2 \text{ and solve for } \mathrm{a_{rgb}(x)}$$

and the ambient

$$Ambient(x) := \frac{1}{size(P)} \sum_{p \in P} I_p(x)$$

4. HIGH-LEVEL OPTIMIZATION

Photometric Stereo needs a large amount of data to produce accurate results. A typical data set might consist of 260 images with a 20 MP resolution, totaling a little over 5 GB. Since the computations will be executed on the GPU, this data will need to be available there. At the time of writing, the TITAN X with 12 GB RAM is the only CUDA-capable GPU that has enough global (DRAM) memory to accommodate this amount of data. The proposed solution should be able to run on any kind of CUDA-capable hardware. Removing this memory constraint is also necessary for images of an arbitrary resolution to be processed, adding to the scalability of the presented solution.

4.1. Slicing

Subdividing the high-resolution images into smaller image slices allows for the circumvention of the memory constraint. This approach is possible since the Photometric Stereo method is pixel-based; there are no inter-dependencies between the pixels, so the full-size images are not necessary to calculate the output for a specific slice. Instead of running the computation on 260 full size images, the computations are run on 260 times a number of smaller image slices. These slices all represent the same portion of their respective full size source images, ensuring that the algorithm operates on the right pixel combinations. The results of these multiple smaller image slices are computed one after the other, producing result images that are themselves slices of the full-size result image. These result slices are recombined at the end of the algorithm, creating the full-size result image. The slices that we used were chosen to be horizontal strips, since these are the fastest to obtain. Diagrams of the image slicing and slice recombination are shown in Figure 5.

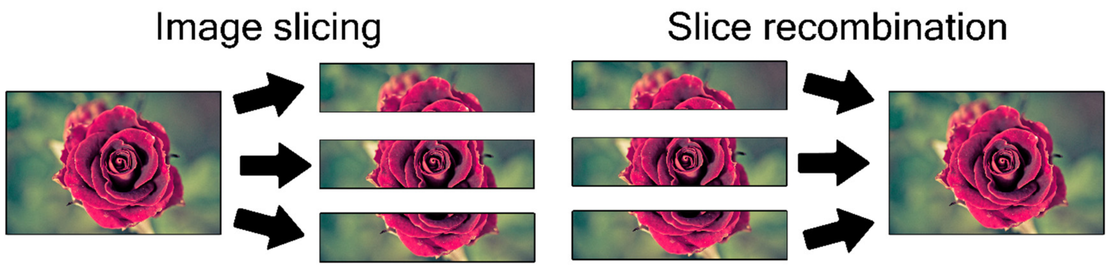

Figure 5: Image slicing diagram (left). Slice recombination diagram (right).

The transfer times from CPU to GPU memory have a profound effect on the overall application performance. Therefore, it is important to ensure these transfers are executed as fast as possible. The data transfer between host and device memory is carried out over the PCI-Express bus (PCIe) for optimal efficiency.

4.2. Pinned memory

Host (CPU) memory is pageable by default. The GPU cannot access data directly from pageable host memory. When a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked host array, then copy the host data to this pinned array, and finally transfer the data from the pinned array to device memory. This process is illustrated in Figure 6.
As shown in the figure, pinned memory is used as a staging area for transfers from the device to the host. The cost of transferring between pageable and pinned host arrays can be avoided by directly allocating the host arrays in pinned memory (Harris, 2012). Pinned memory allows the GPU to use its DMA engine to perform the data transfer without having to involve the CPU. Using only the DMA

engine to perform the memory transfer allows making full use of the PCIe bus bandwidth. Another property of pinned memory is that it is impossible for pinned memory to be swapped out to disk. This behavior is necessary, given that the swap partition is inaccessible to the DMA engines of the GPU. The unpinned memory transfer will be limited by the CPU speed and will be in the range of 1-1.5 GB/s. The pinned memory transfer on the other hand, is limited by the speed of the PCIe bus. For a PCIe 2.0 bus, the transfer speed will fall in the range of 5.5-6.5 GB/s. This effective data transfer rate is lower than the peak bandwidth of 8 GB/s due to interface overhead and other system design trade-offs. For a PCIe 3.0 bus, the effective bandwidth will be around 10-12 GB/s (Eshelman, 2013).

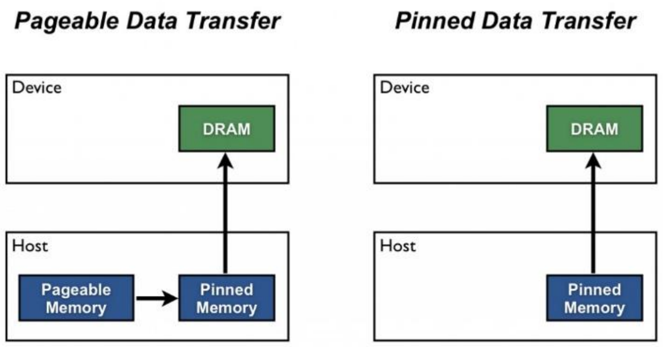

Figure 6: Pageable vs pinned data transfer (Harris, 2012).

4.3. Data transfer concurrency

CUDA provides the option to perform data transfer concurrently with calculations, reducing the amount of stalls due to data dependencies. However, no computations can be done on data that is still being written. To avoid transfer latencies, the device memory is divided into two partitions. While computations are being run on the data in one partition, new data is being written to the other. The partitioning is achieved through the image slicing method previously discussed. Figure 7 shows how one slice is being processed while simultaneously transferring the next one. The size of these image slices is determined by the software to maximize memory occupancy while reducing the latency incurred by the very first memory transfer. If necessary, the amount of device memory used by the application can be limited to allow other processes to run on the GPU as well. A typical case would be using the same GPU to simultaneously render the screen content and to run the computations for an algorithm.

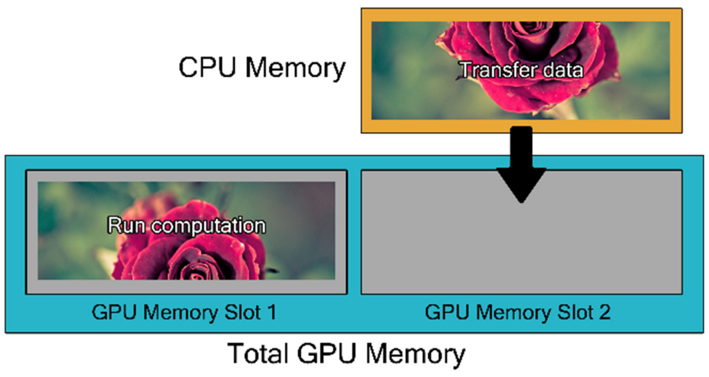

Figure 7: Concurrent memory usage.

4.4. Pipelining and streams

The overlapping of data transfers with both computation on the host and the device is achieved through the use of streams. A stream in CUDA is a sequence of operations that are executed on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to be executed in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently (Harris, 2012). The pipelining schedule is illustrated in Figure 8.

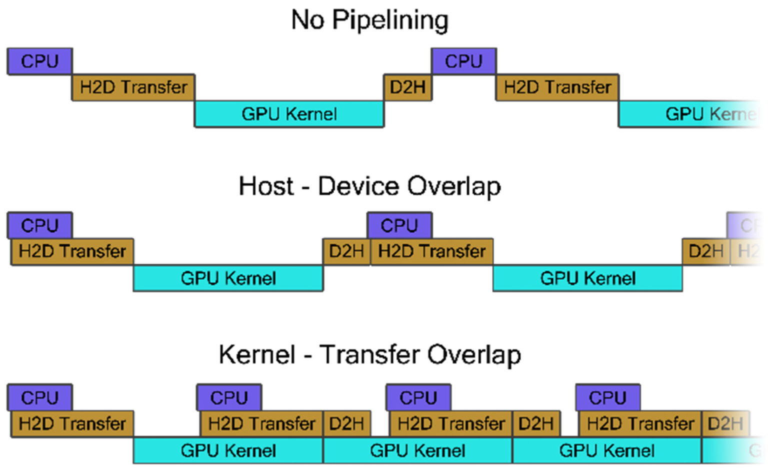

Figure 8: Pipelining schematic.

If all steps are scheduled sequentially, most of the GPU's capabilities are wasted. The GPU will wait for the CPU to finish scheduling tasks and for data transfers to complete, spending most of its time idling. The CPU processing and host-to-device (H2D) transfers of the same cycle can be overlapped, as shown in the diagram. These memory transfers can be overlapped with the execution of the kernel. The GPU is constantly busy with computing new results, without having to waste processing time waiting for new data to arrive. This is shown in the diagram by the continuous GPU Kernel section. Similarly upon completion of some computations, the device-to-host (D2H) transfers of the results can be done simultaneously.
The effective speedup achieved by this pipelining scheme depends on the amount of data that needs to be transferred and the time it takes for the kernel to run (which itself is dependent on the GPU's performance). For the optimal case, where transfer and calculation times are roughly the same, this pipelining can potentially double performance. For the current implementation, a 700 MB slice will take 120 ms to transfer and 132 ms to process. For this test setup, the performance gain through pipelining is $120/(132+120) \approx 47.6\%$.
The pipelining also indicates that there is an upper performance limit. For each 1000 MB of source images, 2076 MB of data needs to travel over the PCIe bus. The presentented implementation is entirely PCI bandwidth limited. Essentially the GPU is processing the data faster than it becomes available. This limit dictates that the maximum processing speed for the implemented photometric stereo algorithm is 3.15 GB/s when using a PCIe 2.0 bus. For a PCIe 3.0 bus, an effective bandwidth of 12.0 GB/s can be reached, increasing the processing limit to 5.78 GB/s. For further performance increases effort must be made to improve the data transfer.

5. LOW-LEVEL OPTIMIZATION

5.1. Overview of the optimization strategies

Creating a massively parallel solution with high performance demands must not be underestimated. Many traditional sequential programming paradigms no longer hold. Poor design and carelessness with respect to architecture are not remedied by branch prediction or prefetching. The following guidelines offer a suggestion as to which strategies tend to be most beneficial.

- Reduce the amount of memory accesses. It is imperative to ensure all memory accesses (both reads and writes) are done coalesced. The data must be properly structured and the data request patterns must be properly organized, so that the number of memory accesses that are needed for the computations can be minimized.
- Reduce register spilling. The right types of memory need to be used to store the right number of variables for faster access times. The overhead of having to access L1 cache needs to be avoided.
- Reduce latency and increase thread occupancy or ILP. The SIMD model needs to be used to maximize the number of active threads while reducing the cost of expensive operations. This is done by keeping the ALU busy at all times. Instruction level parallelism (ILP) allows thread to schedule and reorganize their instructions to reduce the amount of stalls.

Without paying heed to these pillars, any micro-optimization will be negligible. The optimizations should also be attempted in the presented order. Drastically reducing register spilling only to change it subsequently to account for coalesced memory access is meaningless. In the same way, increasing thread throughput is irrelevant if that throughput is bounded by the latency of excessive register spilling. These three strategies are covered in greater depth in the following pages.

5.2. Memory coalescing

5.2.1. Global memory coalescing

Grouping of threads into warps is not only relevant to computation, but also to global memory accesses. The device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. Arrays allocated in device memory are aligned to 256-byte memory segments by the CUDA driver. The device can access global memory via 32-, 64-, or 128-byte transactions that are aligned to their size. Recent CUDA devices (Fermi architecture and up) have an L1 cache in each multiprocessor with a 128-byte line size. Accesses by threads in a warp are coalesced by the device into as few cache lines as possible. If the requested memory is properly aligned to this 128-byte cache line, the requested data segment can be read during a single bus transaction. If the access to this memory is misaligned however, the data request has to be serviced in multiple transactions, spanning multiple cache lines, resulting in additional memory latency (Harris, 2013).
To ensure that the memory accesses of the developed application are properly aligned, each image that is transferred to the device memory is padded so that its size is an integer multiple of 128 bytes. This ensures that the warps can be properly aligned to the cache lines so that superfluous transactions are avoided. This principle is illustrated in Figure 9.

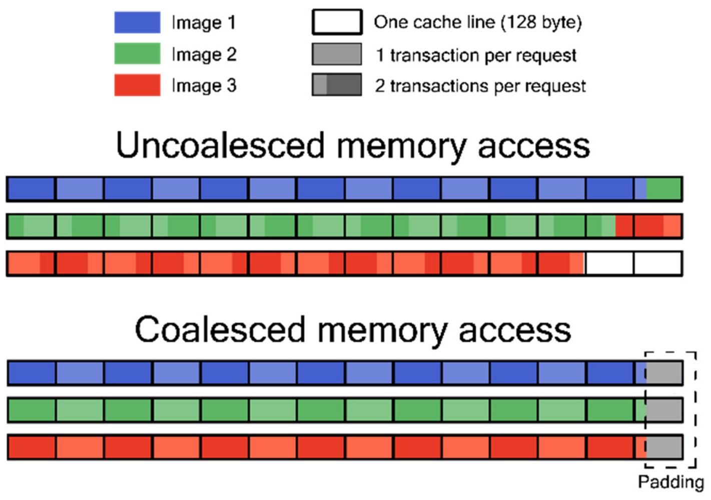
Image 1
Image 2
Image 3
One cache line (128 byte)
1 transaction per request
2 transactions per request
Uncoalesced memory access
Coalesced memory access
Padding

Figure 9: Memory access coalescing.

From the diagram above it is clear that if the image size *S* is not an integer multiple of 128 bytes, a thread block would often have to request device memory loads from two memory banks. The lucky few that are aligned with the underlying architecture enjoy the benefit of only having to do a single load, reducing the bandwidth limited latency by 50%. By padding the original image size to an integer multiple of 128 bytes, every block can access its corresponding pixels from each image with a single load. This is at the expense of *128 – (S mod 128)* bytes. Because photometric stereo is inherently pixel independent, the algorithm does not need to take boundary conditions into account. Instead, it just processes these redundant pixels and subsequently discards them, without any adverse effect for the performance.

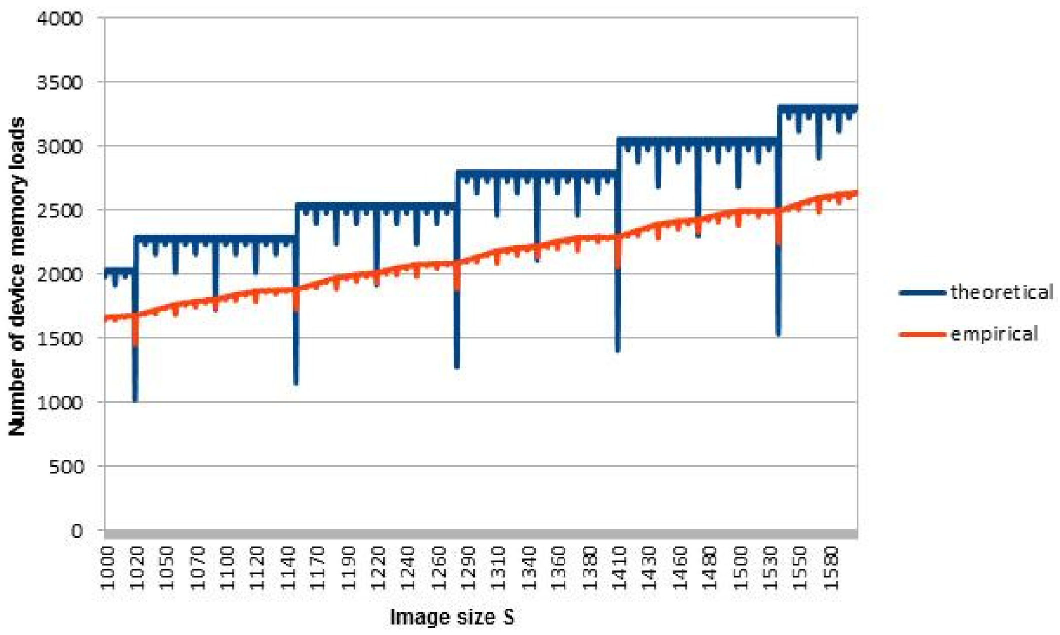

Figure 10: Memory loads needed for images of size *S*.

It is clear that for every image of size *S*, the first image is properly aligned. Also, after a certain number of images *N*, another image is properly aligned. *N* is the smallest number for which *(N * S) mod 128 = 0* holds true. Figure 10 shows the number of memory loads required for image size *S* when

128 threads are requesting consecutive data. For typical data sets of 3 MP, there is a 13% decrease in device memory loads when comparing *S**, a multiple of 128, and *S* - 1*. The large discrepancy between the theoretical and empirical data is evidence of aggressive caching. Each device memory load is checked to see if the value is cached in L1. Upon a cache miss, the request goes to L2 cache and subsequently to DRAM. In the test case when a given block has to issue two load requests, there is a high chance that the result of the first request, which was previously processed by a different block, is cached. The effects of temporal locality amortize the actual number of accesses to DRAM, which is the main cause for such bandwidth limited latency.

5.2.2. Shared memory coalescing

Memory coalescing is equally important for data storage. Albedo and ambient image buffers (single channel) are trivially stored. The normal has an x, y and z component calculated by each thread. Due to the SIMD nature of the architecture, this means that obvious store operations lead to strided accesses, as shown in Figure 11.
Due to strided global writes to device memory, three times as many store operations are scheduled, with a significant performance hit as a result (NVIDIA, 2015). Additionally, multiple entries in L2 cache could potentially lead to data re-fetching for a different thread block. The solution is the introduction of thread cooperation through data reorganization in shared memory, leading to the desired coalesced writes. This is illustrated in Figure 11.

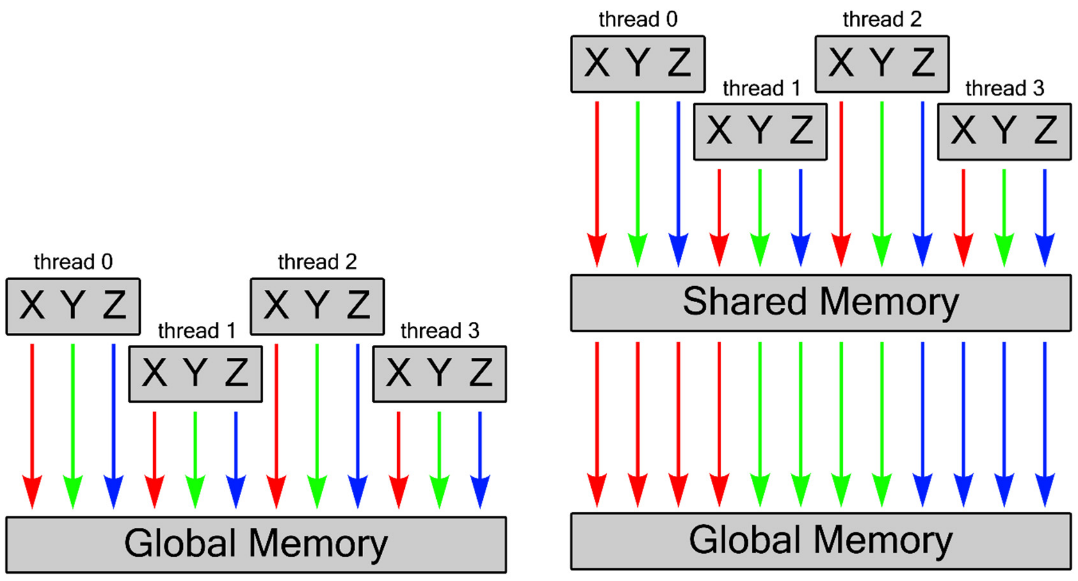

Figure 11: Left: Normal storage without using shared memory, leading to strided access.
Right: Normal storage using shared memory.

Essentially the earlier uncoalesced write to device memory is replaced by an uncoalesced write to shared memory, followed by a coalesced write from shared memory to device memory. Although the indirection through shared memory creates a certain amount of overhead (measured to be 132 clock cycles), this allows for just one single write to device memory. Shared memory as part of the on-chip L1 cache bank allows for a thread to store data that was computed by a different thread. Non-conflicting device memory stores (~400 cycles) are much slower than the cached operations (~38 cycles), resulting in a performance increase of 400 * 3 – (400 + 132) = 668 cycles, essentially reducing the cost by 56%.

The most important aspect of this write sequence is that it is warp independent. This means that each warp operates by itself, without the need to synchronize between the various warps and avoiding the performance hit this would incur. When opting for such a compartmentalized approach, a shared memory size of 12 * 32 bytes per warp is needed. For typical block sizes of 128 threads this translates to 1536 bytes. The scheduling constraints for each SM impose the maximum of 32 concurrent thread blocks (assuming a shared memory size of 48K (NVIDIA, 2015)). This constraint is irrelevant since the number of threads is limited due to the limited resident block total. Effectively this means that the shared memory considerations can never be a limiting factor for thread scheduling. Data reorganization in shared memory is therefore always a good idea for storing the normal results. This holds true for every NVIDIA GPU.

5.3. Register usage

5.3.1. Register spilling

Registers are a precious commodity in NVIDIA GPUs. They are by far the fastest way to locally access and store data. As such, it is often worth copying a variable stored elsewhere into a register before accessing it multiple times. In turn, the number of available registers is very limited. Each temporary or local variable resides in register space, unless there are not enough registers available. In this case, the register values will spill to L1 cache. When using registers, no attention must be paid to memory coalescing or bank conflicts. However, the number of registers used, and when they are accessed, does impact the overall performance. One of the most obvious ways to improve a CUDA program, is to use less registers. Using too many registers could reduce occupancy or even cause spilling to L1 cache. If the variables in a performance critical section (e.g. an inner loop) are accessed from memory outside of register space, this could have devastating effects. However, a little spilling can actually improve performance when the spilling enables a significant increase in occupancy. Older NVIDIA GPUs (CC $\leq$ 3.0) have even less registers than their current counterparts, so even more care must be taken when deciding how many registers to use. Because of the way the hardware and warp scheduler are designed, a thread block must allocate all of its necessary registers prior to executing. The warp scheduler does not take temporal access patterns into account as it cannot guarantee a specific order of executed instructions. The easiest way to reduce register pressure is to only store necessary variables. Variables that can be recalculated cheaply, contain no meaningful data or are only used once or twice do not belong in registers.
For each recorded image, corresponding to a different lighting position, a different light vector L_i is found for each pixel. Light vectors consist of three floats, taking up three registers. These vectors are calculated as the difference between the light source position and the pixel position, and are therefore different for each thread. Due to the large number of light positions for the typical mini-dome (currently 260), each thread would require a total of 780 registers to store them. Obviously, such a large number will not fit in the 255 available registers. The light vectors that do not fit, will spill to L1 cache. If every thread spills 3120 bytes to L1 cache, only 63 threads can run concurrently before inducing further spills to L2 cache, resulting in a maximum occupancy of only 3.1%. Instead of storing these light vectors, it is better to recalculate them when they are actually needed. The current design only uses 3 registers per thread instead of 780. Since this approach repeats the same calculations many times, this actually creates additional overhead. However, this is completely offset by the reduced number of memory accesses and increased occupancy. Simulations show a performance increase of over 3400 times when the light vectors are recalculated, instead of permanently stored.
The most important part of the optimizations concerning register usage, is that they are cumulative. Using one more register could cause another to spill, potentially causing other parts of the program to grind to a halt. Function inlining, compiler and linker optimizations, and instruction level

parallelism (ILP) must all be taken into account when examining the consequences of creating certain variables. Because of this, it is important to analyze the program in its entirety in order to make meaningful predictions about its behavior.

5.3.2. Single vs double precision

When designing any algorithm, serious thought must be given to what type of variables are used to store a given value. Both memory and direct processing speed should be taken into account. Registers are by far the fastest way to access stored data. Each register on an NVIDIA GPU holds exactly one word, which means that storing double precision variables take up twice the amount of memory than simple floats. GPUs are designed to process vast amounts of single precision floats. Each core can process a single precision arithmetic instruction in a single cycle. Conversely a typical GPU, such as the GTX 780 Ti, has fewer ALUs with the capability to perform double operations (8 per SM). Effectively this means that it takes 24 clock cycles to process a 64-bit floating-point instruction (NVIDIA, 2015).
The underlying architecture is best suited to process single precision variables. The downside to using less bits to represent the same number is that an inevitable loss of precision occurs. Before altering the algorithm and eliminating all double precision variables, one must check if it has a noticeable effect on the system output. The first key observation is that, in contrast to the inner operations, the resulting normal, albedo and ambient images are float arrays. Internal instructions introduce rounding errors on intermediate values, which can accumulate and alter the resulting output. These errors are larger for floats than for doubles. To investigate the possible impact of using less significant digits, the output images for both double and float algorithms are compared. If they are deemed close enough, a transition to single precision operations is justified. As a measure of the difference between normal images, the mean Angular Error (AE) measure suggested by Barron *et al.* is used (Barron et al., 1994).

$$mean\,AE = \sum_{i=0}^{S} \frac{AE_i}{S} \qquad with\ S = total\ pixels$$

$$AE = \cos^{-1}\left(\frac{\bar{f} * \bar{d}^T}{\|\bar{f}\| * \|\bar{d}\|}\right)\ with\ f,d = \ [x\ y\ z\ 1]$$

f and d are the real surface normals for pixel i, represented as respectively a float and double. Table 1 presents an overview of the difference in output values when changing from double precision to single precision calculations. The majority of pixels receive an identical normal (AE < 0.001). This comparison indicates that the difference between using float and double is negligible.

Data set	Moth	Clay tablet
Mean AE	0.00181	0.00177
Pixels with AE < 0.001	96.2 %	99.1 %
Mean speedup for float vs. double	5.05 x	5.04 x

Table 1: Overview of differences between double and float.

5.3.3. Packing bytes into words

The data set that is fed to the Photometric Stereo algorithm consists of images where the intensity of each pixel is represented by a single byte. This means that four intensities can be stored per word. Due to the limited resources and significant cost of accessing data, the data should be grouped tightly together. One of the compound types used to hold multiple data is *uchar4* (NVIDIA, 2015). If four bytes are accessed sequentially, four requests might be needed, introducing significant memory latency. Instead, it is usually a better idea to access four consecutive bytes (one word) in a single memory request and store them together in a single register. This register can then be read four times, with better performance. This approach is shown in Figure 12.

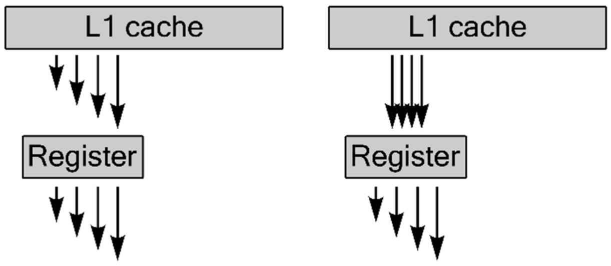

Figure 12: Loading data from L1 cache as four *uchars* (left) or as one *uchar4* (right).

When dealing with the low level optimizations and memory layout, it is often prudent to enforce the usage of certain structures. When CUDA compiles a piece of code, it already applies many optimizations by itself. However, these optimizations are extremely dependent on the context. Small arrays are a classical example of this. CUDA allows these small arrays to be stored in fast register space. To determine the maximum dimension of these arrays, the compiler has to take many factors into account. When the array size exceeds this limit, it gets pushed into L1 cache, as was designed to happen in CUDA. When the optimizer decides not to bundle four bytes into a word, array sizes quadruple. The only way to force the compiler to actually group four bytes together in a single word, is to manually pack them together in a *struct*. Control over the compiler's actions is an often-overlooked aspect of code design. When done correctly, it can save a huge amount of development time. However, this is usually at the expense of more complicated access patterns and reduced legibility.

5.4. Reducing unnecessary operations

From a performance perspective, unnecessary operations are a waste of resources. When optimizing an algorithm, one must first identify the bottlenecks. Reducing a part of your program by 50% is meaningless if the part itself only accounts for 0.5% of the total cost. This consideration is known as Amdahl's law (Amdahl, 1967). It predicts the maximum expected improvement to an overall system, when only one part is improved. The maximum speedup in a program where one part was sped up p times is limited by the inequality (7), where $f\ -(0 < f < 1)\ -$ is the fraction of time spent in the part that was not improved.

$$maximum\ speedup\ \leq\ \frac{p}{1 + f * (p-1)}$$

The normal calculations account for the majority (~85%) of all arithmetic computations. The iterative approach executes the same inner loop for a fixed number of iterations (Willems, et al., 2005). After each step, the predicted normal becomes more accurate, converging to an optimum. This is achieved by reducing the number of equations in the system. The number of steps that is actually required is highly dependent on the characteristics of the surface (such as reflectivity and surface texture). Highly

reflective surfaces for example will produce many saturated pixels reducing the number of valid equations.
This is shown in figure 14. It is easily shown that if two normals of consecutive iterations match, the algorithm has converged. This powerful information was not taken into account in the original CPU implementation.

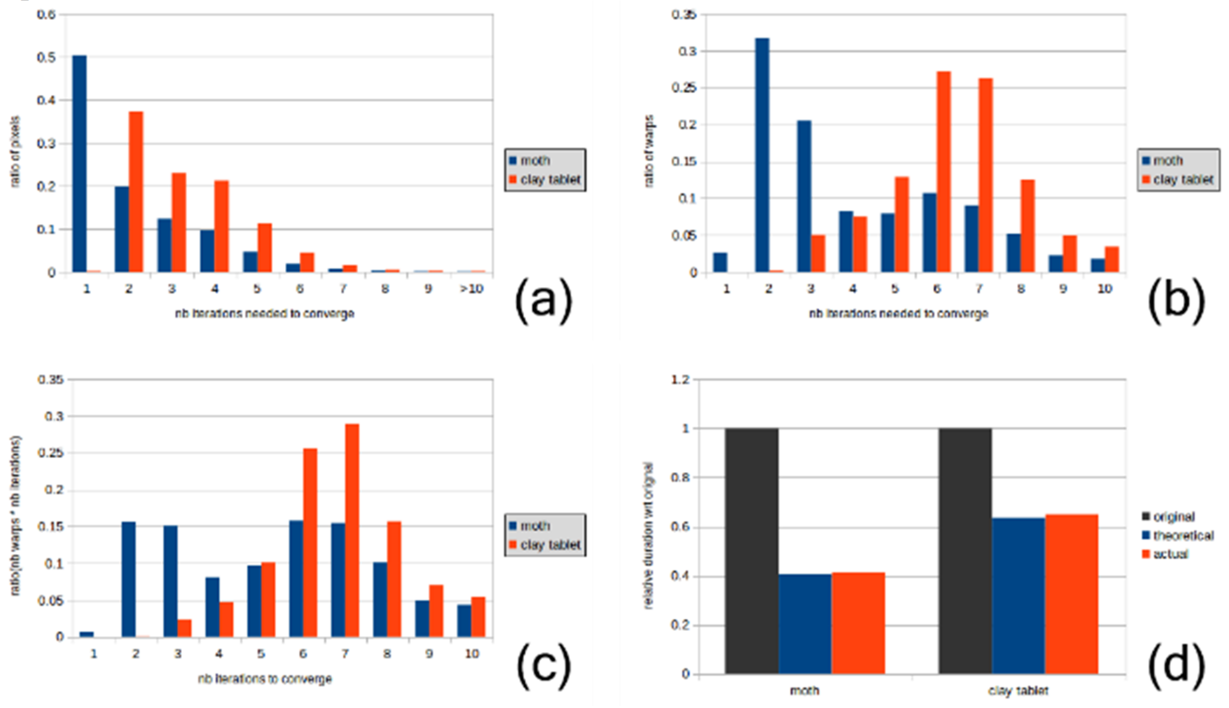

Figure 13: Graphs detailing the number of iterations needed to converge to a reasonable normal for the moth and clay tablet data sets.

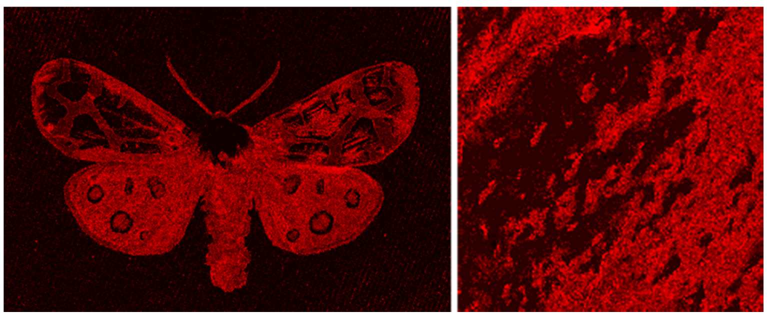

Figure 14: Graphical representation of the number of iterations needed to converge (higher intensity means more iterations).

Convergence is analyzed here for two separate data sets (figure 14). One can examine the number of iterations each thread goes through, before its normal converges (figure 13a). Pixels with either very similar or very dissimilar entries are prime candidates for early convergence. A large number of pixels (98% and 95% of the respective data sets) can be readily computed in 5 iterations or fewer, which is an improvement compared to the fixed value of 10 iterations. However, a thread is never executed by itself. The smallest unit, the warp, has each of its 32 threads execute the very same instruction. This means that if a single thread has not converged, the others will have to wait for it to finish. Performance implications must therefore be analyzed by comparing warps, instead of single threads. A warp has converged if all 32 threads have converged, so the entire warp needs to go through as many iterations as the longest running thread needs.

Warps execute threads that correspond to consecutive pixels. This makes use of the fact that pixels, which are located in close spatial proximity, tend to be strongly correlated. The algorithm itself considers each pixel to be independent of its surroundings, but the spatial locality of the image can be exploited to reduce computation time. Figure 13b shows that over 50% of all warps have finished in 3 and 6 iterations for the respective data sets. Checking for convergence can potentially save all these warps from recalculating known values. An estimation for the number of calculations can be observed by a weighted average of the number of iterations the warps execute. Figure 13c shows how the few warps that do not converge quickly still account for a lot of computations. This can be summarized by taking the total number of iterations for all warps and comparing it with the original requirements where all ten iterations are calculated. Both the theoretical and actual improvements (Figure 13d) are significant considering the high profile nature of this bottleneck.
In order to check for convergence, a copy of the previously estimated normal must be stored and updated during each iteration. The use of three additional registers per thread increases register pressure, but the potential spills to L1 cache are offset by the extra reads from L1 cache that would have occurred if the next iterations had been executed. If the algorithm is performance limited by the total number of registers available in an SM, 3 additional registers could reduce the number of threads that can be run concurrently. This reduced occupancy can potentially negatively impact the performance. A little overhead, from introducing additional branching and the optimization costs that come with it, can account for the discrepancy (1.2%) between the actual and theoretical speedup. By eliminating superfluous calculations and accesses, the computation of the normals can be performed up to 59% quicker for typical data sets. Since these computations represent 85% of the total arithmetic operations, this has a noticeable effect on the overall performance of the algorithm.

6. CONTINUOUS USER FEEDBACK

The performance increase associated with the novel GPU-based implementation of Photometric Stereo fundamentally changes the way the Minidome can be used. Previously the high computation times forced researchers to collect their datasets, store them and only later process them. This is no longer the case as the real-time solution can provide the user with the desired output upon completion of the recording. The recording itself has become more time consuming than the calculations. Unfortunately many measurements are not optimal due to bad lighting and camera configuration or object placement. Previously the researcher would have to wait until the completion of the recording before deeming it a success or simply making an adjustment to the camera settings and restarting the process.
The presented implementation allows for continuous user feedback because of shorter computation times. Specifically the algorithm can be started with a subset of the final 260 images corresponding to 260 different light positions. While the resulting normal, albedo and ambient images are not as accurate they still give a clear indication as to whether the recording itself will be a success. As more images are collected the quality of the resulting images increases and converges towards the eventual outputs. The algorithm was designed to be easily optimized for different number of images and as such can be run concurrently with the recording software.
Each time a new image becomes available it is transferred to the GPU and the image is taken into account during the next iteration of the algorithm. The resulting images of the partial solution need not be sent back to the CPU and stored. Instead they are visualized using OpenGL. The interoperation between CUDA calculations and OpenGL rendering happens through common memory buffers. By correctly setting up the OpenGL contexts and post-processing the results from Photometric Stereo can be visualized when told to refresh. From a design perspective the online operation used for real-time visual feedback changes little about how the algorithm is optimized. The rendering itself, using

the GPU, incurs a slight performance hit, but that is irrelevant when compared with the advantages that feedback to the user has.
The feedback between the researcher and the recording software allows for premature aborting if the user determines the setup to be suboptimal. The recording times depend greatly on the lighting and

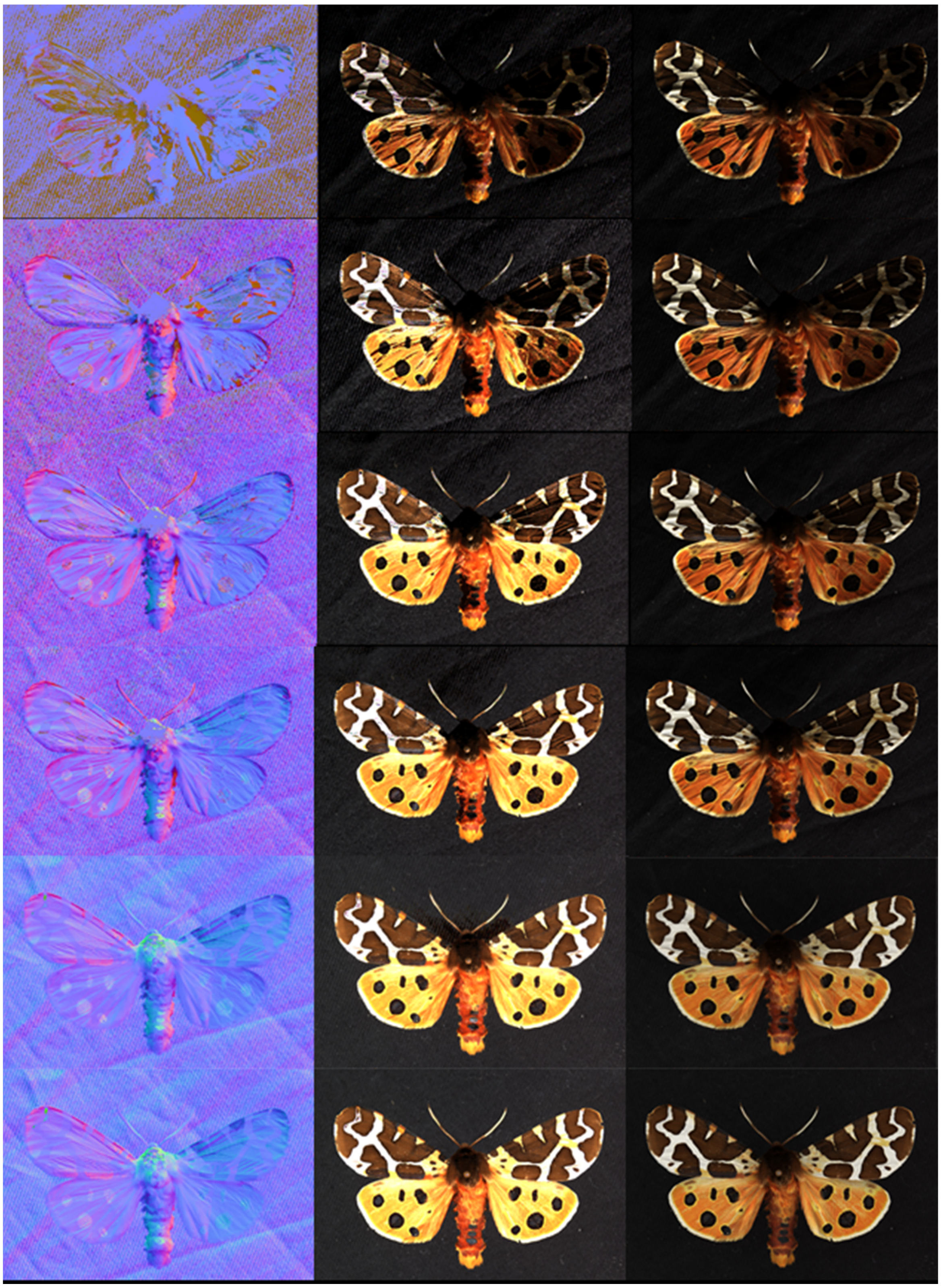

Figure 15: Continuous feedback. Outcome of Photometric Stereo for an increasing number of images. From top to bottom N={10,20,40,60,160,260}. From left to right: normal, albedo, ambient.

camera configurations, with 1.25 images/s being a typical rate for the Minidome. For a full measurement (N=260) over 3 minutes are needed. The online visualizer allows the researchers to work much more efficiently. Moreover this allows them to stop recording when enough data is collected. If the resulting image at (N=200) is of sufficient quality and it is determined that the additional 60 images would not contribute greatly to the result, the measurement can be concluded early. Figure 15 shows an example of this convergence. The first images give an idea of the expected result and after N=160 images the resulting normal, albedo and ambient have almost completely converged. The visual feedback provides additional value for the Minidome and the way it is used.

7. RESULTS

Significant performance increase has been observed compared to the original serial CPU implementation. The proposed solution is still heavily dependent on the hardware it is run on, and was optimized specifically for devices such as NVIDIA's GTX 780ti. The implementation automatically adapts to various image sizes and scales well as depicted in table 2.

	CPU	GPU
1.2MP	158*10^3	265
1.7MP	252*10^3	267
2.7MP	401*10^3	431
3MP	445*10^3	480
6MP	891*10^3	966
28MP	4185*10^3	5009

Table 2: total computation time in ms for GPU implementation and the CPU implementation it was based on, for different image resolutions. (CPU=Intel core i7 2.80GHz; GPU=GTX 780ti).

The novel implementation allows for image processing that was previously too slow and not very time efficient. The ability to scale well with increasing image resolution means that the Minidome

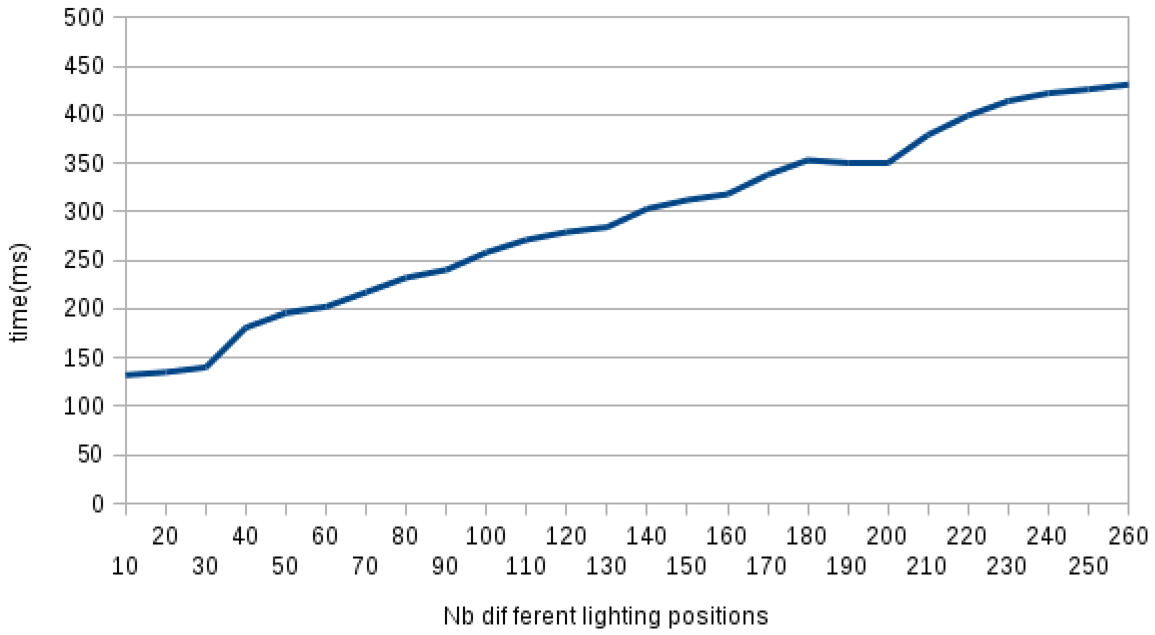

Figure 16: total computation time for a subset of the 2.7MP moth data set. The number of images used to run Photometric Stereo on is greatly dependent on the number of images in the data set.

can be used as a versatile recording device, for many different applications. The number of light positions taken into account is the single most important factor when determining performance for Photometric Stereo. Essentially the implementation presented in this paper was designed to process input data sets of N=260 images. However in online operation mode, with user feedback, fewer images are processed which leads to greatly reduced computation times. Figure 25 depicts the relation between increasing N and the performance cost.

Figure 16 takes into account the time taken to copy the data to and from the GPU. When providing live feedback the program keeps the images locally, removing the necessity for many image transfers and further reducing the performance costs. The result is fast, high quality normal, albedo and ambient images that give an indication of the geometry of the observed object. These results can further be used to create depth maps and render photorealistic 3D models. Result images for three separate data sets are depicted in figure 17.

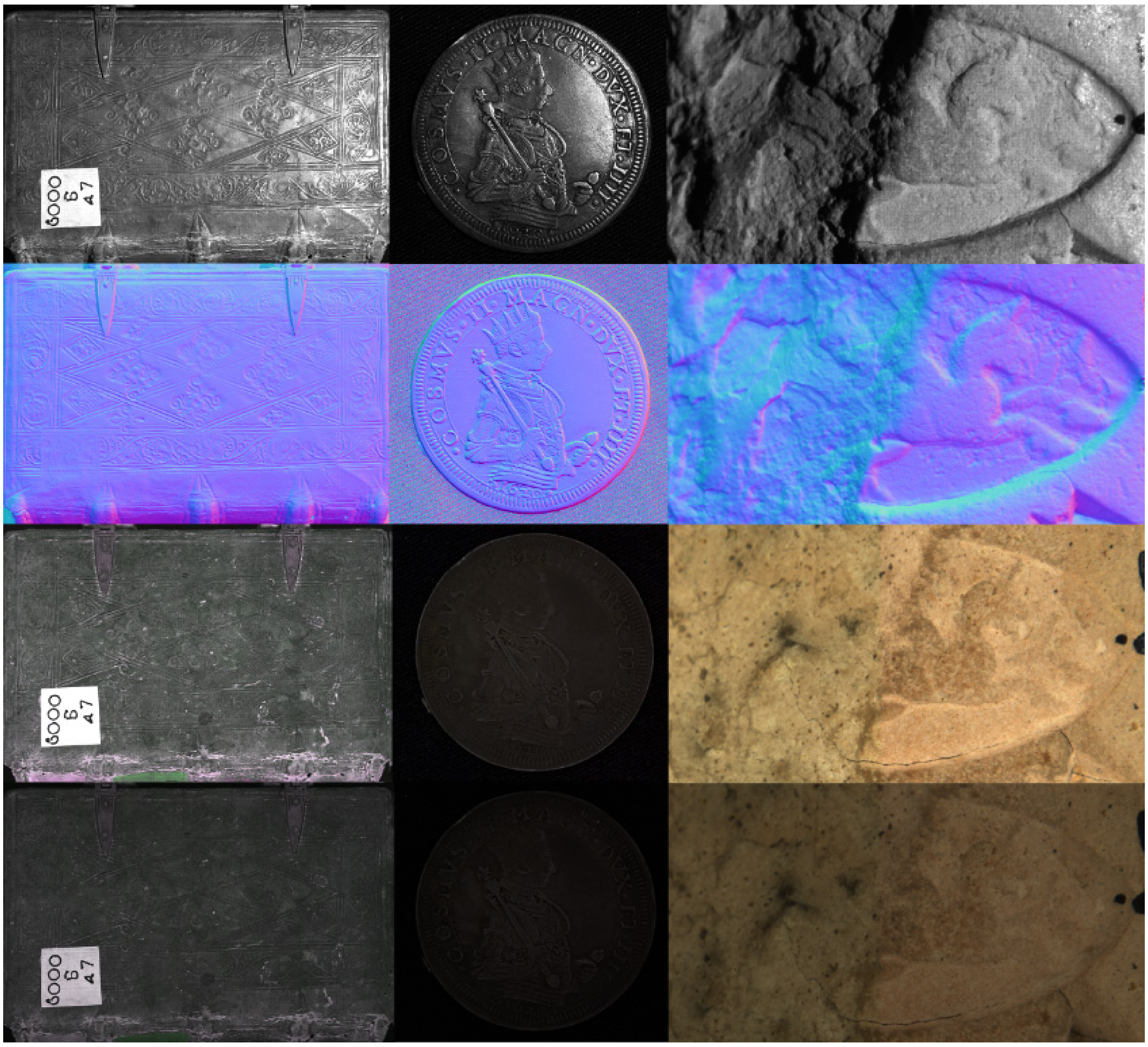

Figure 17: Result images for 3 separate data sets: book(28MP), coin(2,6MP), tablet(1.7MP). From top to bottom: one of the input images observed with a specific lighting condition, normal, albedo, ambient.

8. FUTURE WORK

8.1. Algorithm extension

The current implementation of the photometric stereo algorithm uses regular images as its input, captured in the visible region of the electromagnetic spectrum. Instead of these, the input can be extended to use multispectral images. Research is currently being done, at KULeuven and elsewhere, on using these multispectral inputs for photometric stereo applications (Nam & Kim, 2014). Using the multispectral reflectance information allows for the removal of interreflection on diffuse materials, resulting in more accurate surface normals. The implementation can also be extended to work on non-Lambertian surfaces using a kind of BRDF model to better observe specular reflectance. A multi-camera, multi-light system can be significantly accelerated by scanning BRDF functions, allowing for further optimization.

Another extension is adapting the software to process a variable number of images. Not all 260 images are needed to calculate accurate results. A smart combination of complementary lighting positions could greatly reduce the number of images needed. By running a prediction algorithm on the intermediate results, the light position that is most likely to contain new information can be determined. Using this technique would allow for obtaining an equally accurate result, but fewer images would need to be captured and processed. This enables a reduction in both processing and recording time. For the algorithm implementation, this means that for each pixel, not only the intensity needs to be known, but also to which light position the intensity belongs. In the current implementation, the light position can be derived from the location in memory. Keeping track of these light positions has considerable implications for the software design. On the other hand, the high speed of calculations would render it possible to develop strategies for on-line light source selection, thus bringing further accelerations by reducing the total number of images to be processed.

8.2. Reducing data transfer

The current implementation uses a double input set; both the bayer images and the debayered grey images are transferred over the PCIe bus. Since the current implementation is not yet limited by the bandwidth of the PCIe bus, this is not problematic. However, there is a possibility that the algorithm will become bandwidth limited when more powerful hardware is available, or when new optimizations are applied. The necessary bandwidth can be halved by only sending the bayer images over the bus. This adaptation allows the processing speed to be raised from 3.15 GB/s to 6.3 GB/s. To get the grey value information, the debayering would have to be done on the fly on the GPU. Additionally the amount of data could be compressed to further reduce the total amount of data being transferred.

9. REFERENCES

Amdahl, G. M. (1967). Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. Proceedings of the AFIPS Spring Joint Computer Conference, (pp. 483-485). IBM Sunnyvale, California.

Barron, J., Fleet, D., & Beauchemin, S. (1994). Performance of optical flow techniques. International Journal of Computer Vision, pp. 43-77.

Bibikov, S., Fursov, V., Nikonorov, A., & Yakimov, P. (2011). Memory Access Optimization in Recurrent Image Processing Algorithms with CUDA. Pattern Recognition and Image Analysis, pp. 377-380.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., & Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. Journal of parallel and distributed computing, 68 (10), pp. 1370-1380.

Eshelman, E. (2013, November 18). NVIDIA Tesla K40 “Atlas” GPU Accelerator (Kepler GK110b) Up Close. Retrieved from Microway: https://www.microway.com/hpc-tech-tips/nvidia-tesla-k40-atlas-gpu-accelerator-kepler-gk110b-up-close/

Harris, M. (2012, December 4). How to Optimize Data Transfers in CUDA C/C++. Retrieved from Parallel Forall: http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

Harris, M. (2013, January 7). How to Access Global Memory Efficiently in CUDA C/C++ Kernels. Retrieved from Parallel Forall: http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

Karimi, K., Dickson, N. G., & Hamze, F. (n.d.). A Performance Comparison of CUDA and OpenCL. Burnaby, British Columbia, Canada, V5C 6G9: D-Wave Systems Inc.

Malzbender, T., Wilburn, B., Gelb, D., & Ambrisco, B. (2006). Surface Enhancement Using Real-time Photometric Stereo and Reflectance Transformation. Rendering Techniques, *2006*, 17th.

Nam, G., & Kim, M. H. (2014). Multispectral Photometric Stereo for Acquiring High-Fidelity Surface Normals. IEEE Computer Graphics and Applications, pp. 57-68.

Nozick, V. (2010). Pyramidal normal map integration for real-time photometric stereo. In EAM Mechatronics *2010* (pp. 128-132).

NC State University (2015). Albedo. Retrieved from Climate Education for K-12: https://www.nc-climate.ncsu.edu/edu/k12/.albedo

NVIDIA (2015). CUDA C Programming Guide. NVIDIA Corporation. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/

NVIDIA Compute (2009). PTX: Parallel Thread Execution ISA Version 1.4. NVIDIA.

NVIDIA Corporation (2014). NVIDIA GeForce GTX 980 Whitepaper. NVIDIA.

Patel, H. (2010). GPU Accelerated Real Time Polarimetric Image Processing throug the use of CUDA. Aerospace and Electronics Conference (NAECON), Proceedings of the IEEE 2010 National (pp. 177-180). Fairborn, OH: IEEE.

PCI-SIG (2015). PCI Express® 4.0 Frequently Asked Questions. Retrieved from PCI-SIG: https://www.pcisig.com/news_room/faqs/FAQ_PCI_Express_4.0

Poynton, C. (2012). Digital video and HD: Algorithms and Interfaces. Elsevier.

Remenyi, A. (2011). Biomedical image processing with GPGPU using CUDA. MIPRO 2011. Opatija, Croatia.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (pp. 73-82). New York: ACM New York.

Schindler, G. (2008, June). Photometric stereo via computer screen lighting for real-time surface reconstruction. In: Proc. International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT), CD-ROM.

Song, G., & Biao, G. (2012). Research on Multi-GPUs Image Porcessing Acceleration Based CUDA. International Conference on Industrial Control and Electronics Engineering (pp. 196-199). Xi'an Technological University, China: IEEE.

Varnavas, A., Argyriou, V., Ng, J., & Bharath, A. A. (2010). Dense Photometric Stereo Reconstruction on many Core GPUs. Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on (pp. 59-65). San Francisco, CA: IEEE.

VISICS. (2015). Retrieved from Minidome: www.minidome.be

Volkov, V. (2010, September 22). Better Performance at Lower Occupancy. Retrieved from UC Berkeley: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

Watteeuw, L., Vandermeulen, B., Van der Stock, J., Truyen, F., Proesmans, M., Van Gool, L., & Gradmann, S. (2014). Imaging the topography of illuminations and bookbindings with reflectance transformation imaging. ICOM-CC 17th Triennial Conference. Building Strong Culture through Conservation. Melbourne, 15-19, Abstract No. 543.

Watteeuw, L., Vandermeulen, L., Van der Stock, J., Delsaerdt, P., Gradmann, P., Truyen, F., Proesmans, M. Moreau, W., & Van Gool, L. (2013). Imaging Characteristics of Graphic Materials with the Minidome (RICH). ICOM-CC Graphic Documents Working Group Interim Meeting, Vienna 17-19 April 2013, pp. 140-141.

Willems, G., Verbiest, F., Moreau, W., Hameeuw, H., Van Lerberghe, K., & Van Gool, L. (2005). Easy and cost-effective cuneiform digitizing. The 6th International Symposium on Virtual Reality, Archaeology and Cultural Heritage. VAST: The Eurographics Association.

Yanqing, F., Fu, L., Danwei, G., Xu, L., & Xiaoling, L. (2012, March). The processing of palmprint image based on CUDA. In: Automatic Control and Artificial Intelligence (ACAI 2012), International Conference on (pp. 2162-2166). IET.