|  |  |
|---|---|
| Project acronym: LADIO<br><br>Project number: 731970<br><br>Work package:<br><br>Deliverable number and name:<br><br>D2.3: File format implementations verified in LADIO prototypes | Title: Data Model and API<br><br>Work Package: WP2<br><br>Version: 1<br>Date: August 31, 2017<br><br>Author:<br>Carsten Griwodz |
| Type:<br>[ ] Report<br>[ ] Demonstrator, pilot, prototype<br>[ ] Website, patent filings, videos, etc.<br>[X] Other | Co-Author(s):<br>Michael Polic, Jonas Markussen<br><br>To:<br>Albert Gauthier, Project Officer |
| Status:<br>[ ] Draft<br>[ ] To be reviewed<br>[ ] Proposal<br>[X] Final / Released to EC | Confidentiality:<br>[X] PU – Public<br>[ ] CO – Confidential<br>[ ] CL - Classified |
| Revision:<br>Final | |
| Contents:<br>Deliverable 2.3. File format implementations verified in LADIO prototypes. | |

# Table of contents

# Introduction

Deliverable 2.3 is a software deliverable that provides the database and file formats that have been implemented in LADIO according to plans.

OpenMVG is capable of reading a variety of video formats, which can either be processed as images or as a contiguous sequence of frames. 360 videos and images have been integrated into OpenMVG in two ways, one that splits them into 6 separate image sequences for the 6 faces of a cube, and one that processes equirectangular frames directly. Some more information is given in 360 video and image support.

The inclusion of LIDAR data has yet not advanced beyond the theoretical state and the documentation from Deliverable 2.2. The practical examples that are part of the first LADIO dataset[1] are packet traces in the PCAP format. Although we can convert them into point clouds using the the PCL library's HDL Grabber[2] both in the LAS format that was chosen in Deliverable 2.1 and into Alembic files for use by CMPMVS, we rely on Task 3.1 (finalized in Deliverable 3.1, month 15) to truly integrate this LIDAR data. Some more information is given in LIDAR support.

Deliverable 3.3 deals with the creation of accuracy information in image-based 3D reconstruction, which has so far received little attention in practical systems. The algorithm that has been newly developed in LADIO for computing the covariance matrices for large numbers of cameras and points (and thus derive uncertainty information) based on a (first) reconstruction is described there. It makes the use of accuracy for large-scale recordings feasible for the first time, but integration is not complete and the storage is preliminary in the form of text files. Some more information is given in Accuracy information.

The performance enhancements for processing speed in 3D reconstruction are frequently limited by available main memory, and for processing steps that are off-loaded to the GPU, there are limitations to the transfer speed. For Intel-based Linux machines, we bypass the CPU step and allow subsequent stages of GPU processing to communicate directly with SSDs. More explanations about this are given in section Direct GPU IO.

## 360 video and image support

360 videos are usually encoded in the rather inefficient equirectangular format (see the discussion in Deliverable 2.2). LADIO partners are currently using the Ricoh Theta, which is relatively cheap and easily accessible.

The Theta has 2 fisheye cameras that are mounted back-to-back. It performs real-streaming of two fisheye video streams packed side-by-side into a single H.264-encoded frame and performs offline stitching and recording of H.264-encoded video at 4K in an equirectangular format. Whereas the stitching gives quite good frame-by-frame results, huge changes in alignment between frames can occur. In general we can conclude after our experiences that the Ricoh Theta does not deliver sufficient quality for the final state of LADIO. Its recording quality is quite low and it software stitching leads to unpredictable

[1] http://ladioproject.eu/#datasets
[2] http://pointclouds.org/documentation/tutorials/hdl_grabber.php

distortion. For the time being, however, it is feasible to continue using it due to its low price and because it gives us the opportunity to work on different 360 camera image layouts as well as introducing static, predefined camera Rigs into the openMVG stage of our reconstruction pipeline.

The existing code is experimental. In the openMVG branch dev_rig[3], preliminary code loads frames in dual fisheye mode and introduces static rig information into the reconstruction pipeline. The input can be both individual frames or videos. The rig information is stored in Alembic files and can later be loaded from them. The rigs are integrated with both the incremental and global structure-from-motion pipelines of openMVG. Most work is based on the side-by-side fisheye input from the Ricoh Theta, which does not introduce the distortion of the software stitching.

In the openMVG branch popart_develop_de265[4], 360 degree videos that are either encoded in equirectangular format or in CubeMap format are divided in 6 frame sequences, where each sequence represents one side of a cube with the pinhole camera at its center. Figures 1 and 2 show a frame of an equirectangular 4K video split into the 6 faces of the same frame in CubeMap projection.

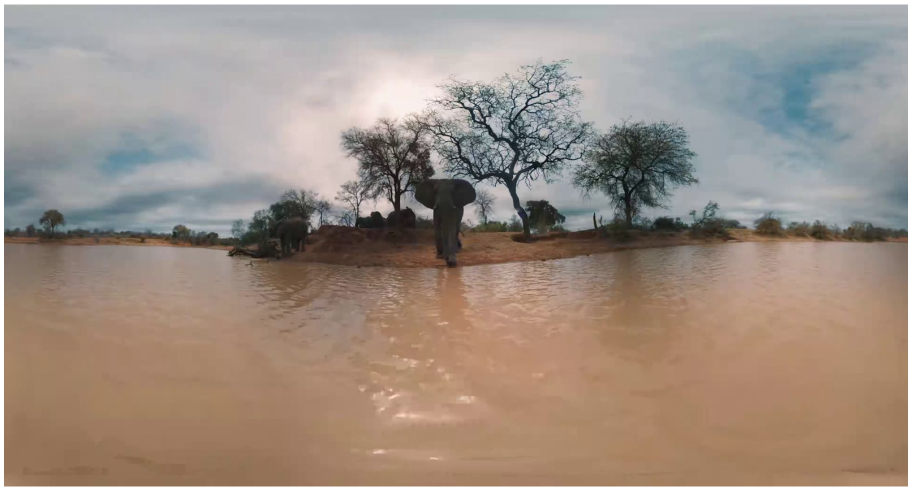


Figure 1: equirectangular frame

[3] https://github.com/alicevision/openMVG/tree/dev_rig
[4] https://github.com/alicevision/openMVG/tree/popart_develop_de265

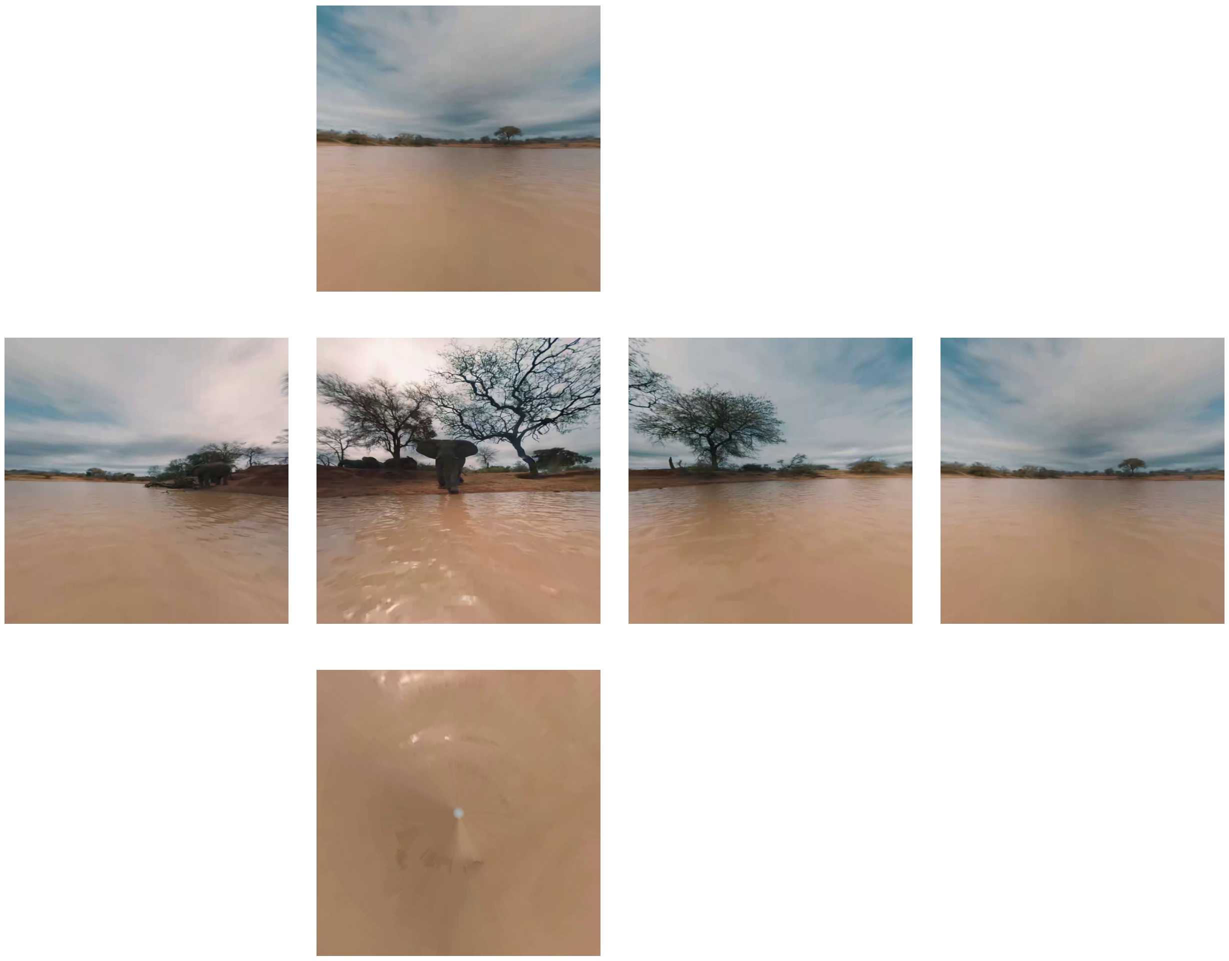


Figure 2: Cube surfaces of a CubeMap projection

## LIDAR support

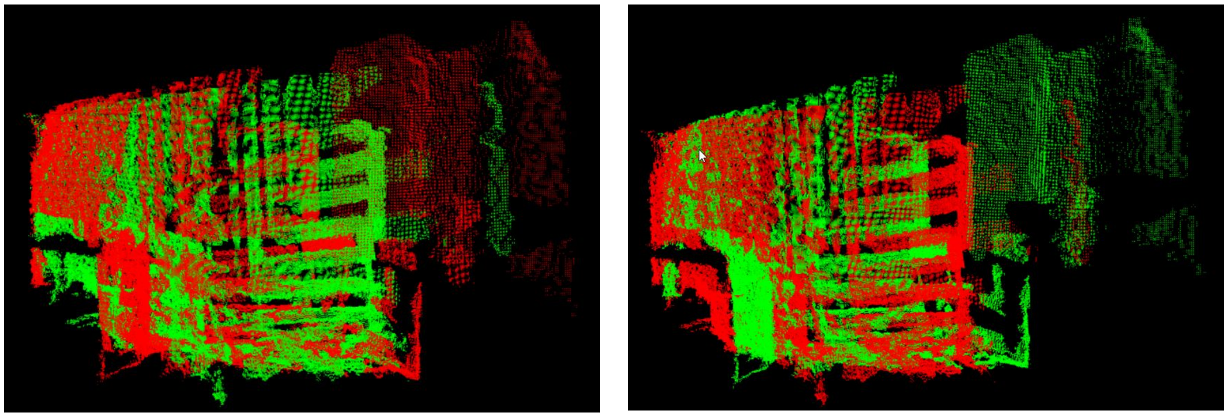
The current LIDAR data that is available in the project is delivered from Velodyne LIDAR scanners, which are frequently supported by companies that provide scanning services to film sets. The price of LIDAR scanners is still too high for acquiring a decent-quality LIDAR scanner as part of the regular film equipment.

In contrast to expectations (and advertising), LIDAR scanners stemming that have been developed to support autonomous vehicles are not feasible for 3D reconstruction at this time. These scanners operate in real-time for the requirements of autonomous vehicles, which require the immediate detection of objects that are present within a so-called sector surrounding the car. Once such objects are detected, other means of understanding these objects are employed, typically based on image-based machine learning systems. Due to this specific demand, current implementations deliver high-frequency section scans that cover large areas, ie. the LADIO interpretation would be that of huge pixels. While more reasonable pixel densities are promised for future (cheap) LIDAR scanners derived from those for the autonomous vehicle industry, they do not actually exist yet. Consequently, LADIO work does currently relate to the more expensive Velodyne scanners.

The output of the Velodyne scanners, however, is not a standardized format. Quite the opposite, the only way of capturing Velodyne output is in the form of network-level packet traces that must be interpreted using the network capturing library libpcap[5] or a libraries that fulfill the same task. So far in LADIO, we have only had use for data delivered in the VLP-16 format that is generated by the Velodyne Puck.

For integration into LADIO, a higher-level interface that does already provide an interpretation of the libpcap-parsed data packets exists. It is part of the Point Cloud Library (PCL)[6], a BSD-3-licensed software, and is called HDL Grabber.

The amount of information that can be found in these packets provides only a subset of the information that is specified in the LAS format, but it exceeds the information that we expected from cheaper LIDAR scanners. Each Velodyne Record of 1200 bytes contains scans at 16 different elevations (-15° to +15°) and a reported rotation angle. For every scan, strongest and last returns are reported, and data includes RGB color information. The timecode of every scan is based on a time reported by an optional GPS add-on module (note that the GPS module was not operational for all samples in the LADIO dataset[7]).



Two frames captured from LiDAR scanning, before and after alignment

## Accuracy information

Making use of accuracy information in the reconstruction pipeline has been identified as an opportunity for improving first, the decision about inclusion of new frames during incremental SfM, and second, the weighting of matched points during camera tracking. The algorithms for this are the topic of Deliverable 3.3, but we are also storing this information.

So far, the generated data file stores the information in human-readable format. Based on reconstruction information generated by OpenMVG, the camera parameters, covariance matrix for each camera, and 6-value covariance matrix for each reconstructed point are stored in a text file. The code is found in a

[5] http://www.tcpdump.org/
[6] http://pointclouds.org/
[7] http://ladioproject.eu/#datasets

*private* AliceVision repository[8]. The code has recently been integrated with OpenMVG, and we are now ready to take the integration a step further and make use of the uncertainty information in OpenMVG's incremental SfM and camera tracking.

## Direct GPU IO

For memory intensive tasks, such as 3D reconstruction, processing speed can be severely limited by the size of the available memory. When memory is exhausted, the least recently used data must be stored on disk in order for more recently used data to be loaded into memory. For programs running on the CPU, this is handled automatically by the operating system, where old data is swapped out to disk automatically and loaded back in when needed. For programs running on a GPU, however, the programmer must keep track of where memory is and must manually move data between RAM and onboard GPU memory. In addition, the programmer must also handle disk IO using a CPU thread in order to read disk blocks into system RAM and write them back to disk, as well as keeping this process synchronised with threads running on the GPU.

Figure 3a and Figure 3b illustrate this process. For simplification reasons, we assume that blocks are already mapped out and that the appropriate IO commands are already prepared in memory. (1) Synchronization point between GPU threads and CPU thread; (2) CPU thread allocates a memory buffer and triggers the disk controller, indicating that there is a IO command ready for it in the IO submission queue and blocks; (3) the disk controller fetches the IO command from the queue; (4) the disk controller reads disk blocks into a memory buffer; (5) the controller indicates to the blocked CPU thread that the IO commands are completed; (6) the CPU thread initiates a memory transfer; (7) the GPU's onboard DMA engine reads from system memory into GPU memory.

[8] https://github.com/alicevision/uncertainty

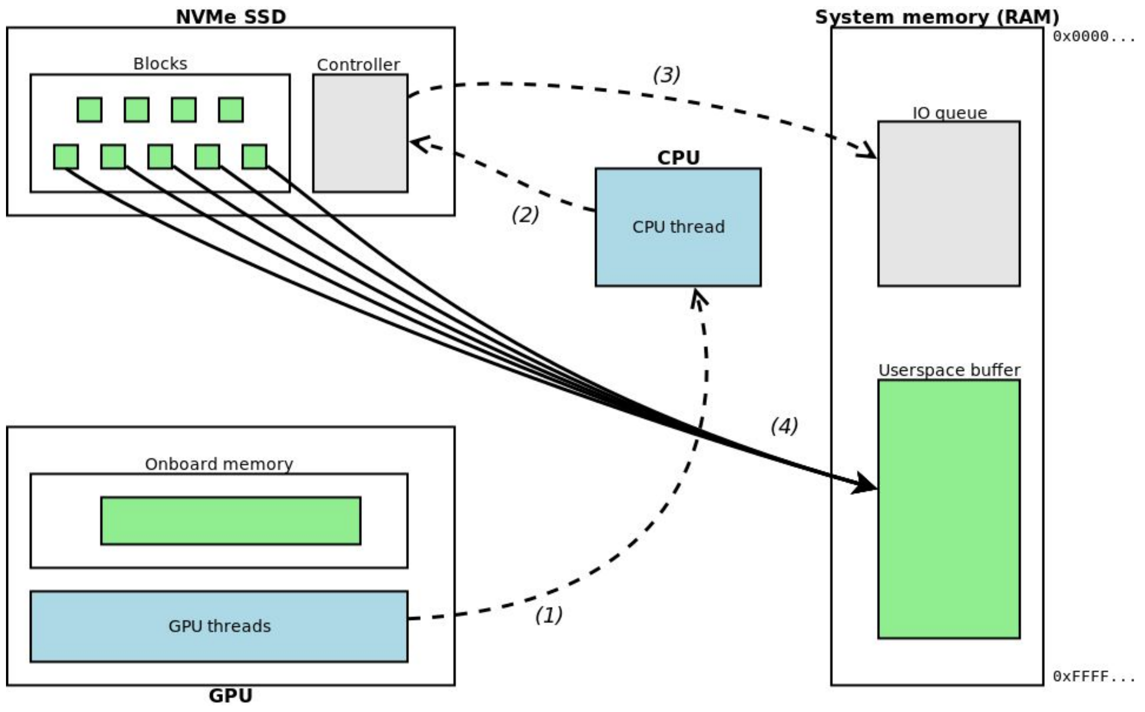


Figure 3a: Disk IO using a CPU thread

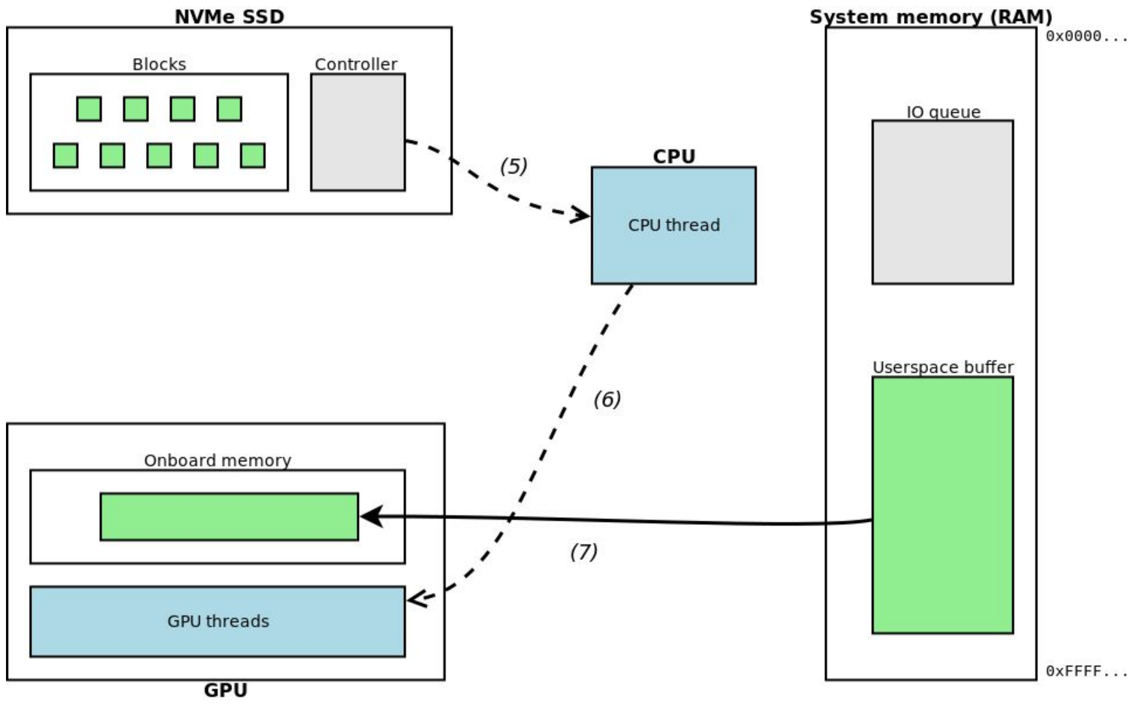


Figure 3b: Disk IO using CPU thread (continued)

It is possible to simplify the process of disk IO for GPU programs by exploiting modern SSDs that support NVM Express (NVMe)[9]. NVMe is a disk controller specification for non-volatile storage media attached on the PCIe bus. It is designed to be fully asynchronous, reflecting the parallelism of current multi-CPU architectures, and therefore supports lock-free command queues and completion signalling. It also specifies that the disk must be able to read and write into any valid address, effectively allowing data to be read from and written anywhere in the entire address space.

Using the workstation models of Nvidia GPUs, it is possible to expose onboard GPU memory to third-party PCIe devices using the GPUDirect API[10]. As the GPU and the SSD are both attached to the same PCIe bus, and share the same address space, it is possible for the SSD to read from and write directly into onboard GPU memory, rather than having to be bounced through system RAM first. Since NVMe is fully asynchronous, the disk can be controlled entirely from the GPU without needing a CPU thread at all.

This approach, however, has some considerations. Firstly, bypassing the file system abstraction entirely and operating directly on disk block level means that file formats must be converted from on-disk representation into a format that is suitable for direct processing by the GPU. This is trivial to implement, as unpacking image and video file formats is already handled by the application and can be easily extended into a pre-processing process. Secondly, both the GPUDirect API as well as setting up disk mappings require low-level driver support. While this is achievable on all operating systems, it is currently only implemented for Linux. In other words, this acceleration is currently only possible on Linux machines using workstation variants of Nvidia brand GPUs.

[9] http://www.nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf
[10] http://docs.nvidia.com/cuda/gpudirect-rdma/index.html

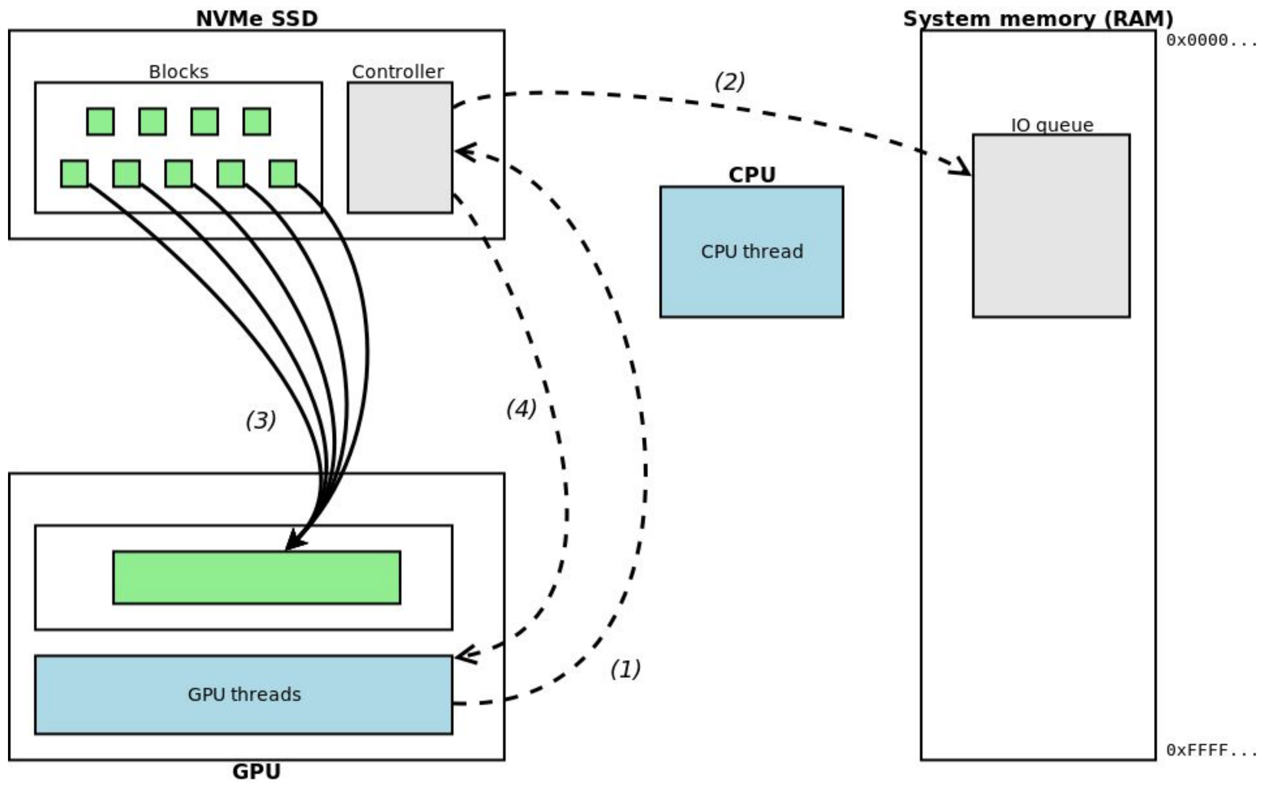


Figure 4: Direct disk IO initiated by a GPU

Figure 4 demonstrates direct disk access from the GPU. Again it is assumed that data to block mapping is already prepared in advance for simplification. (1) The GPU indicates to the disk controller that IO commands is ready for it in the IO submission queue (in system memory); (2) the controller fetches the IO command from the queue; (3) the disk controller transfer disk blocks directly into GPU memory; (4) the disk controller notifies the GPU threads that disk IO is complete. Note that it is not necessary to host the IO queues in main memory, they may be hosted in GPU memory entirely.

Direct GPU IO acceleration is currently in a stand-alone Github repository[11], and will be integrated into CMPVMS once the CUDA implementation moves forward.

[11] https://github.com/enfiskutensykkel/ssd-gpu-dma