THE "PHOTOGRAMMETRIC LOAD CHAIN" FOR ADS IMAGE DATA AN INTEGRAL APPROACH TO IMAGE CORRECTION AND RECTIFICATION

M. Downey [a, 1], U. Tempelmann [b]

[a]Pixelgrammetry Inc., suite 212, 5438 – 11 Street NE, Calgary, Alberta, T2E 7E9, Canada
michael.downey@pixelgrammetry.com
[b]Leica Geosystems AG, Heinrich-Wild-Strasse, 9435 Heerbrugg, Switzerland
udo.tempelmann@leica-geosystems.com

Theme Session, ThS-4

KEY WORDS: Imagery, Software, Processing, Performance, Rectification, Visualization, Radiometric, Automation

ABSTRACT:

A new photogrammetric load chain has been developed for the ADS40 pushbroom sensor focusing on parallel computation and interactive image processing. It allows image correction and rectification at load time, useful as well for interactive work for the generation of image products. The performance needed to fulfill these goals has only come with the move to multicore general purpose computers and programmable graphics cards. A flexible framework is needed that encourages data parallel design. A programming model called "stream processing" has been the approach taken by many when developing parallel libraries. In this paper we explore the stream processing model and how it applies to an ADS40 work flow. We look at how the model will apply to both general purpose computers and programmable graphics cards.

1. .INTRODUCTION

Early photogrammetric and remote sensing image processing consisted of a sequence of separate processing steps; with each step taking data from a storage device, processing, and returning the data to the storage device. The visualization of the result was a separate step, which did nothing more than to draw the image to an output device. With the introduction of interactive workstations, it became clear that the visualization would benefit from real-time corrections of the current image data (image patch). This inspired the idea of a configurable "image load chain", which applies single pixel and neighbourhood operations (such as contrast stretch or image sharpening) "on the fly" whenever an image patch is loaded. The same load chain is also useful for operations on complete images by making separate pre-processing operations obsolete, with the additional benefit of reduced storage space. Higher-level operations such as image rectification, while chainable, have not been considered for interactive work at this time. They have however, been introduced into specific processing steps such as image matching.

At the time the first version of the Leica Geosystems Airborne Digital Scanner (ADS40) processing flow (GPro) was created, contemporary computing power made it impractical to set up a complete load chain. The biggest problem was the complexity of the sensor model for an airborne pushbroom sensor (with its short term orientation variations). Consequently two product levels were created on disk: "L1" (plane-rectified) for stereoscopic work and "L2" (ortho-rectified) as an intermediate product for ortho-mosaics and remote sensing applications. A radiometric load chain was implemented into the rectifiers and the viewer.

The main goal of a new ADS processing package was to eliminate the need of saving intermediate results and to apply a completely configurable set of low- and high-level image corrections on-the-fly. For example, a complete load chain for stereo-viewing RGB (Red, Green, Blue) or FCIR(False Colour Infra-red) L1 would apply the following processing steps separately for each stereo partner:

- decompress image patch into an input cache
- plane-rectify patch to a three-band image
- apply local automatic or user-defined radiometric stretch
- put into output cache.

In a more complex example, such as the production of a four-band remote sensing ortho-image, the processing steps could look like this:

- decompress image patch into an input cache
- apply additional sensor calibration
- convert to an at-sensor radiance image
- apply model-based atmospheric correction
- convert to a ground reflectance image
- apply model-based BRDF (Bidirectional Reflectance Distribution Function) correction
- ortho-rectify to four band image
- put into output cache.

Most load chain steps require parameters. These must be either supplied (e.g. sensor calibration), extracted from the flight data (e.g. exterior orientation) or extracted from the image in a pre-processing step (e.g. radiometric statistics).

[1] Corresponding author.

One critical point in creating products from digital high resolution images is the processing time. This performance problem can be partially overcome by using modern multi-core CPUs and multi-CPU computers. Further improvements in throughput can also be achieved by distributing tasks in a computing cluster.

All the more critical is the performance in a viewer application - where the latency from patch request to patch display has to be minimal. Even with the use of multi-core/multi-CPU solutions, the processing latency can not be reduced to a reasonable value if rectification is included in the load chain. Massively higher performance can only be achieved with special acceleration hardware. Although many hardware solutions are cost prohibitive, the use of advanced consumer-grade GPUs (Graphics Processing Units – aka video cards) has the optimal mix of high computing performance and low cost.

2. DEFINITION

Throughout this paper, the expression "load chain" is used for a special image processing chain, with the following characteristics:

- the processing is driven from the output side by a request for an image region and a processing level
- the load chain determines the input area and the processing kernels to apply
- input data is buffered in one or more *input cache*s
- processing is done by one or more *kernel*s
- kernels are either chained directly or use a cache layer in-between
- the output data is placed in an output cache, from where the requesting procedure gets the result

3. STREAM PROCESSING

In the past few years consumer level processing units have moved from a single all powerful unit to multiple units. The reason has been the inability to continue the exponential leaps in processing power using a single processing unit. This change is causing software to become more parallel aware, and is changing the approach software developers take when solving a problem.

One way to solve the parallel programming problem is to use a stream processing model. In the stream processing model complex programs, called kernels, operate on a collection of data sets or streams. These streams are organized in such a way that any number of kernels can execute in parallel on different areas of the streams. The key goal of the stream model is to limit the need for a large amount of synchronization between each running kernel, which allows for efficient use of the processing hardware. The forerunner to the stream programming model is the Single Instruction Multiple Data (SIMD) model. In this model a single instruction is performed on a large set of data all at once. Most modern CPUs provide some sort of SIMD instruction set, yet compilers have difficulty using the instructions because of the way current software is designed. The move to a stream processing model allows for both easier use of SIMD instructions and more specialized hardware such as GPUs and FPGAs (**F**ield-**P**rogrammable **G**ate **A**rrays).

In the stream processing model a large amount of data is first loaded into a fast storage area and a kernel is then called on each element in that data. The maximum number of kernels as possible are run to efficiently use the processing units provided in the hardware. One of the big advantages of the design is that it allows for good scaling to newer hardware (as adding more processing units will allow more kernels to run).

The stream processing model fits nicely with how GPUs work. Most GPU architectures have a large number of processing units combined with a high speed memory system. Each processing unit is designed to run the same program, which can be loaded and unloaded before and after each run. The incoming data is broken into equal chunks and passed to each of the units to process. For example, the NVIDIA 8800GTX contains 16 multiprocessing units each containing 8 simple processors for a total of 128 processing units. Up to 6 GPUs can be attached to a single motherboard, giving the GPU system a large amount of processing power when compared to current CPU architectures (which are at four cores and two sockets per motherboard for consumer grade hardware).

The disadvantage of the GPU architecture is that it does not have direct access to data from disk and relies on the CPU system to load new data into its main memory. This data load is an extra set up cost that is ideally amortized by the processing advantages that GPUs have over CPU architectures. An additional drawback with this architecture is that most GPU programming APIs have been designed around graphics programming, and general purpose programming has historically needed to be fit into the graphics programming model. Recently a number of newer APIs have been developed with general purpose computing in mind. Both NVIDIA (with their CUDA programming language) and AMD/ATI (with their CTM programming language) have enabled an easier fit for general purpose programming.

With the recent movement of CPUs to more and more processing cores, the stream processing model is now quite usable on a pure CPU platform. The difficulty now is to provide programmers with a way to take advantage of the more parallel architecture. OpenMP was created to help alleviate this difficulty of writing parallel algorithms on multipurpose CPUs. OpenMP provides the management framework required for running a program on multiple processing units. The programmer now only needs to organize the data in the correct way to allow for good parallelism.

4. LOAD CHAIN

From the beginning of our load chain design we had decided to follow the stream processing model to take advantage of GPUs as well as multicore CPUs. We separated the design into three main areas: caches, kernels, and control. Caches would be the backbone of the system providing high speed access to the data. Kernels would be the computational components which could run on both CPU and GPU systems. The control component would be comprised of a management layer for synchronization and an I/O layer which would allow for offloading file reads.
The structure of the load chain follows a fairly simple design where data moves into a cache, gets processed by a set of kernels, and then is written into another cache. An output cache can have any number of input caches, but each input cache can only have at most one output cache. This restriction

is needed to reduce the amount of synchronization that would be needed between caches.

To support both a CPU and GPU based load chain we needed to define caches that work on either system. Since GPUs rely on the CPU for data, GPU caches can only load data from other CPU or GPU caches. CPU caches have no restriction and can be loaded from a file as well as a CPU or GPU cache. Transfers between CPU and GPU caches need to be minimized as those transfers are relatively slow. Once data is transferred onto the GPU it should remain there until the final result is computed.

Kernels are the computational components of the load chain and perform a single operation on data in a cache. Any number of kernels can be added to a cache and the kernels will be run in series starting with the first kernel added to the cache. Each kernel will be run on as many processing units that are available. Once all kernels have run on a specific cache, control is passed up to the next cache in the chain and its kernels are then run. This continues until all caches in the chain are processed and the resulting data is passed to the user.

5. CACHING

Considering that the stream processing model is centred on the management of data, a well designed caching system will be critical for the success of any processing library. The main design goal for the caching system will be to maximize data throughput. We will require quick access to any data in the cache and access to that data will need to be thread safe. An asynchronous update mechanism will also be needed to allow any portion of the cache to be loaded from disk while a separate area is being processed. Two main caching methods are being considered: 'least recently used' and 'toroidal'.

The 'least recent used' (LRU) cache is a well known design where the cache uses the last access time for an area to determine whether that area should be kept in the cache. Areas that are accessed often will generally be kept in the cache while areas that aren't accessed frequently will be removed. In some situations LRU caches provide good performance, but they have a number of problems. Neighbouring areas in image space are not necessarily neighbouring in the cache, so access to image neighbours can be slow. Additionally most image manipulation kernels will be fairly consistent in accessing all the data in the image the same number of times. All image areas therefore have similar priority in the cache, and will generally be used and then removed from the cache removing most of the benefits of the LRU design.

Toroidal caches take a different approach in that they provide good localized accessed to a specific area of an image. Toroidal caches could be viewed as a double ring buffer where the address of a specific pixel in the cache wraps based on the modulus of the dimensions of the cache. So for a cache with dimensions of 1024 wide and 1024 high, a pixel at line 25000 and sample 1320 would be at row 424 (25000 modulus 1024) and column 296 (1320 modulus 1024) in the cache. As the cache is moved, data at the boundaries of the cache are replaced with new data from disk. When the cache is move to the right, new data is loaded on the left, and when the cache is moved down new data is loaded at the top. The advantages of the toroidal cache are that it provides good neighbourhood access, since all data is stored contiguously in the cache, and neighbouring pixels are generally located next to the current pixel. Since memory areas are fixed, data loads into and out of the cache are well optimized. Toroidal caches also fit very well into GPU architectures, as GPUs provide hardware support for wrapping address modes.

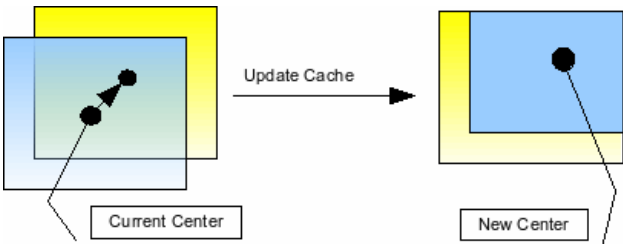

Figure 1: Moving a toroidal cache up and to the right

The other key to providing a high speed caching system is having a way to asynchronously update the cache. The approach we have taken is to have all disk reads preformed in a separate thread. The cache will send messages to this thread to have certain sections of the cache loaded. Once the loader thread completes its work it signals to the load chain that the new data is ready for processing. With an intelligent read ahead algorithm the next data to be processed is being loaded while the current data is processed. This keeps the load chain from stalling, which provides large performance benefits. Table 1 shows the results for reading a compressed image that is 12000 pixels wide and 45336 lines high. All tests were performed on a Intel Core2 Duo 6420.

Test	Real Time	CPU Time used
No Read ahead	20.391 seconds	16.629 seconds
Read ahead	19.885 seconds	16.529 seconds
No read ahead, image processing	28.965 seconds	25.934 seconds
Read ahead, image processing	20.217 seconds	25.874 seconds

Table 1: Read ahead performance tests

With no image processing applied, the times with and without read ahead are practically the same. But once an image processing algorithm is applied to the data we see a big difference when read ahead is enabled. We will always be limited by the time it takes to read new data from the image, but with enough processing cores we should be able to perform many processing steps on the data without being limited by the processing time.

6. KERNELS

A kernel is described as the set of code that you want to run on a specific cache. Kernels need to be thread safe and fairly simple. Kernels should be given a single output pixel to create and be allowed to access any number of input sources. The reason for making kernels very simple is to allow a large number of them to run in parallel without the need for large amounts of synchronization.

A simple example would be a kernel that scales all pixels in an image by a factor of 2. The kernel itself would be given an image location to work on (row, sample) and would then first read that data, multiply the value by 2, and then write the value back into the same location. The program using the load chain would then iterate over the entire image by requesting the load chain to read a tile at a time causing the load chain to run the kernel on every pixel.

Another example would be where we would want to display radiometrically corrected, rectified images in a viewer. The viewer would require a load chain that first decompressed the incoming L0 data, ran a radiometric correction on that decompressed data, and then rectified the result into an output cache. The load chain would be comprised of an input cache for the compressed input data combined with a decompression kernel that would decompress the data. The next step would require a cache that contained the decompressed data and a radiometric kernel that would perform the radiometric correction. The last step would be comprised of the output cache and a rectifier kernel which would take the radiometrically corrected data and generate rectified image data. The viewer would be able to position the output cache at any location. The load chain would determine how to position the input caches to generate the resulting data. Once the input data is loaded the kernels would run in sequence and the output cache would be filled for that area. The load chain would also attempt to determine the next area to be requested and load that data into the input cache as well.

The number of kernels running concurrently would be based on the number of computational units available. Therefore the total speed of the processing should scale linearly to the number of computational units.

By using the stream processing model we can properly optimize certain libraries that have historically been limited to a single processing unit. An example would be a library which reads compressed image data. In the simplest case that library would read a chunk of data from a file, decompress it, and then pass that data up to the calling system. The problem with this approach is that the decompression isn't set up to run in parallel with the reading of data. With decompression being handled as a kernel in the load chain we are able to see large improvements in the total time taken to read a compressed image.

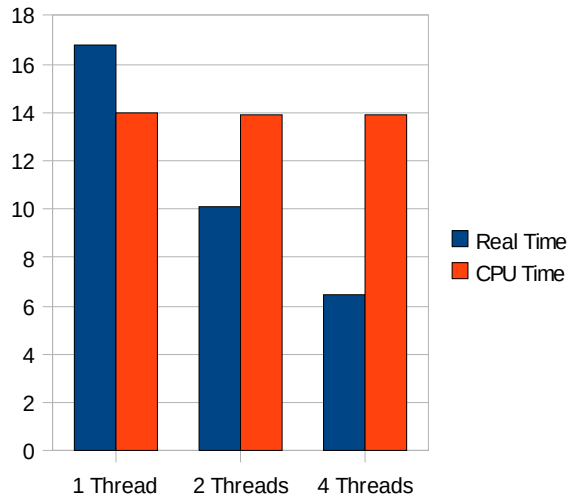

Figure 2: Decompression performance

As the number of processing units increases the real time taken to do the load and decompression of an image decreases dramatically.

The simple approach of running the kernel on every pixel generally doesn't work well when applied to current CPU architectures. The high overhead of creating a large number of threads and the context switching of those threads will greatly degrade the performance. More work needs to be given to each thread so that the overhead of creating the thread is minimized and the need to switch between threads is greatly reduced. Most parallel programming APIs don't force that each kernel must only work on one pixel at a time so there are easy ways to fix this issue.

OpenMP is a parallel programming API that permits an easy way to control how a processing task is divided up among a number of processing units. OpenMP provides a number of preprocessor commands that can be added to C and FORTRAN code to tell the compiler how to parallelize a certain area of code. For example if you wanted to multiply each pixel by a factor of 2 you would write something like this in C:

```
for (i = 0; i < height; ++i)
{
    for (j = 0; j < width; ++j)
    {
            data[i][j] *= 2;
    }
}
```

This would run on a single processing unit and would take time proportional to the size of the image to complete. The same code using OpenMP would be:

```
#pragma omp parallel for
for (i = 0; i < height; ++i)
{
    for(j = 0; j < width; ++j)
    {
            data[i][j] *= 2;
    }
}
```

The difference is minimal between the version with OpenMP and the version without; but now the time taken to perform the computation will be divided by the number of processing units available. With OpenMP the definition of a kernel is changed in that it now receives the total area that needs to be worked on and it is up to the kernel to determine how to parallelize the computation.

The GPU approach is similar - and there are a number of programming APIs which can be used. Currently we have focused on using NVIDIA's CUDA programming library. CUDA requires the programmer to break the problem area into a grid of blocks where each block has a fixed number of threads. Each thread can then run a given program or kernel. Grids and blocks can be of one, two, or three dimensions. The thread is given it's location in the block as well as the block's location in the grid. The thread then determines both its input and output addresses based on its address in the grid of blocks. The advantage of the GPU architecture is that the cost of thread creation and context switching is low, so running thousands of threads has minimal impact on the overall performance of the

run. This means that kernels can be relatively simple in that they only need to focus on outputting a single pixel.

The GPU has some disadvantages which make using it difficult. Currently no consumer level GPU supports 64 bit floating point formats natively. Only 32 bit floating point is supported. CUDA currently downgrades all requests for 64 bit floating point to 32 bit floating point. This is a problem for computations that need high accuracy, like ortho-rectification. Newer versions of GPUs are planned to support 64 bit natively so this issue should be solved soon.

The other issue with GPUs is that they require proper memory alignment and well planned memory access to get the best performance. This is an issue in the CPU architecture as well, but GPUs tend to be affected more by this problem. In general however, memory alignment and memory access can be fixed reasonably easily.

Even with the drawbacks described, the GPU provides a large computational advantage over CPU systems. The simpler approach allows for more processing units to be built on the chip which allows for more kernels to be run in parallel.

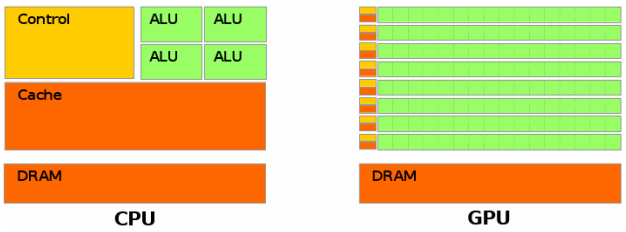

Figure 3: The GPU Devotes More Transistors to Data Processing. (CUDA Programming guide © NVIDIA Corporation 2007)

Figure 3 shows the different focuses of both the GPU and CPU architectures in a very general sense. Current CPU architectures have historically focused on cache sizes, with caches becoming a large portion of the CPU. Numerous examples using GPUs report speed ups of 30 times that of a CPU based implementation. Algorithms that fit well into the GPU system are ones that have a high arithmetic intensity. Ortho-rectification should be a good candidate for running on the GPU once GPUs support 64 bit floating point formats.

7. CONCLUSIONS

With the movement of computing systems towards a more parallel architecture and with the introduction of newer, more specialized parallel systems there is a great need to build software around a parallel framework. In this paper we have described one way to build an image processing chain so that it will take advantage of the increased computational power of multicore systems.

The focus around a stream processing model has allowed us to provide a simple yet powerful framework that can be used throughout our software. We have been able to see significant gains with the approach in both data throughput and computational speed.

8. FUTURE WORK

Our main goal with the load chain will be to provide rectification on the fly of radiometric corrected images. The input data would be compressed L0 images which will need to go through decompression, radiometric correction, and rectification before being sent to a viewer. With this approach we would alleviate the need for creating L1 images before viewing.

Once GPU systems support 64 bit floating point formats we will want to add support for rectification using GPU systems to gain even more performance over current multicore systems. Eventually the entire load chain could be run on the GPU which would minimize the costs associated with transferring data between the GPU and the CPU.

REFERENCES

Gummaraju, J. and Rosenblum, M., 2007. Stream Programming on General Purpose Processors. *38th Annual International Symposium on Microarchitecture(MICRO-38)*, Barcelona, Spain

http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf (29 Nov. 2007)

http://www.openmp.org/mp-documents/spec25.pdf (May 2005)

NVIDIA, 2007. NVIDIA CUDA Compute Unified Device Architecture Programming Guide version 1.1.

OpenMP Architecture Review Board, 2005. OpenMP Application Program Interface version 2.5.