General Purpose Computing on Graphics Processing Units: Decomposition Strategy

Henry Au, Gregory Lum
Space and Naval Warfare Systems Center Pacific (SSC Pacific), Pearl City, HI
{henry.au, gregory.lum}@navy.mil

Abstract

This paper describes the optimization strategies when porting traditional C/C++ algorithms which run on CPU's to parallel processing architectures found on Graphics Processing Units (GPUs). The CUDA parallel programming architecture is also explored through the use of NVIDIA's Visual Profiler for performance analysis. Real time video feeds, such as from onshore surveillance cameras, offer limited visibility when fog, haze, smoke, or dust clouds are present. In order to enhance the video, image processing algorithms such as the Adaptive Linear Filter (ALF) are performed. However, algorithms such as the ALF require large computational time thus limiting the picture quality, size of the video, or number of video feeds being processed concurrently in real time. The GPUs parallel processing computational power is exploited to attain speed ups so that image processing can be performed on the fly in real time. Thus, surveillance is enhanced by providing visual improvement for detection and classification of objects in low-visibility conditions using the ALF. The ALF was selected to provide an image processing context for algorithm optimization on GPUs. The optimization strategies being explored will be CUDA Host memory allocations, streams, and asynchronous memory transfers. Performance results of the ALF running on the GPU and the GPU after optimization will also be reported. As well, GPU limitations will also be briefly discussed in this paper as not every algorithm will benefit from execution on parallel processing architectures.

1. Introduction

Graphics Processing Units (GPUs) have been in use in one form or another to display information to users since the 1980's. GPUs continued to evolve from simple shape accelerators to performing more complex computations such as 3D rendering. However, only as recently as 2007 did General Purpose Computing on Graphics Processing Units (GPGPU) become a viable option for high performance computing. This availability is due to NVIDIA's Compute Unified Device Architecture (CUDA). CUDA has provided a lot of the back end coordination required for managing the hundreds of parallel cores found on their GPUs. As well, an added benefit of GPGPU is the ease with which GPUs can be added or upgraded to a standalone desktop machine for increased performance.

Using the large number of cores available on a single GPU, a desktop computer or even laptop can become a mobile HPC device. This makes it ideal for military applications where mobility, package size, and energy requirements are important factors. Remote drones or unmanned aerial vehicles (UAV) suddenly become possible applications. With a GPU installed on a UAV, data can be processed in near real-time on the aircraft instead of post processed at a remote site when time sensitive information is required.

Figure 1 shows how a set of data is segmented and processed in parallel using blocks and threads. A thread is a set of operations that processes data independent of order, thus allowing for parallel execution. Multiple threads create a block and multiple GPU cores process multiple blocks at the same time. With this architecture it can be easily seen that additional cores results in more data being processed in parallel. Thus, overall computational time is reduced. This makes it more efficient than a CPU that processes data sequentially. However, there are limitations associated with GPGPU due to the fundamental differences between CPU and GPU cores. The CPU core is much more robust and faster enabling it to handle a wider range of tasks when compared to a GPU core. However, since CPU processors have orders of magnitude fewer cores than GPUs, when dealing with highly parallel computations the GPU outperforms the CPU in floating point operations per second (FLOPS). As will be discussed later GPGPUs do have limitations.

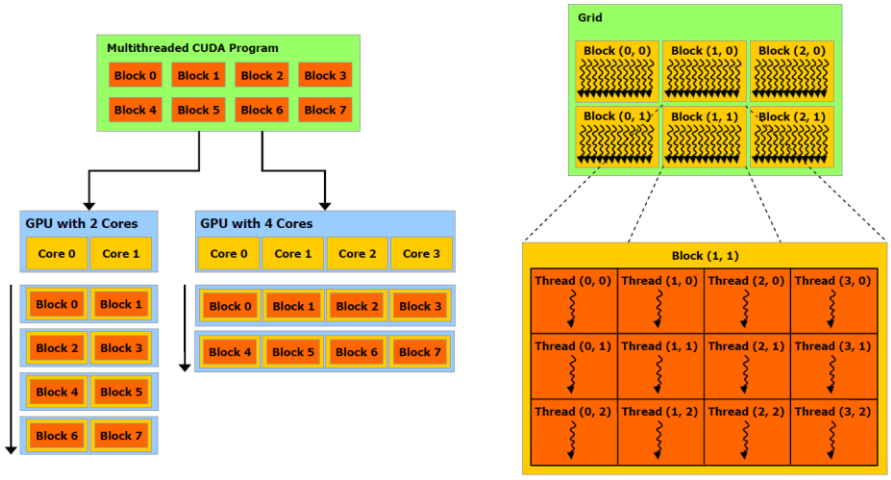

Figure 1. Parallel computation of data

2. Methodology

In order to utilize the computational power of GPUs a standalone desktop was built with the following hardware: an Intel Core i7, 8GB of DDR3 RAM, 7200 RPM hard drive, 1000W power supply, two NVIDIA GTX 460 graphics cards (one for computer display and the other dedicated for processing). Software utilized included Windows 7 64bit Pro, CUDA Toolkit 4.2, University of Oregon's Tuning and Analysis Utilities (TAU) 2.20 timing software, NVIDIA's Visual Profiler, and Microsoft Visual Studio 2008 Professional. Figures 2 describes the CUDA enabled NVIDIA GTX 460 graphics card used for parallel processing. It should be noted that NVIDIA offers much higher performance GPUs dedicated for general purpose computing, such as the Fermi-based GPGPUs. The GTX 460 was selected due to its lower price point. It can be seen that the GTX 460 supports concurrent copy and execute with 1 copy engine. Other enterprise level cards have 2 copy engines allowing for greater host to device and device to host memory copying performance.

```
Device 0: "GeForce GTX 460"
  CUDA Driver Version / Runtime Version          4.2 / 4.2
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1024 MBytes (1073414144 bytes)
  ( 7) Multiprocessors x ( 48) CUDA Cores/MP:    336 CUDA Cores
  GPU Clock rate:                                1440 MHz (1.44 GHz)
  Memory Clock rate:                             1800 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Figure 2. NVIDIA GTX 460 GPU Specifications

The ALF algorithm was profiled using TAU 2.20, with special consideration taken to ensure that functions which occur repeatedly, such as *for loops*, were profiled. The timing information from the ALF TAU profile was then used to develop a decomposition strategy. It is important to keep in mind that profiling a system that currently operates at near real time through CPU processing may not reveal bottlenecks based only on timing. It is therefore important to understand the algorithm being profiled as

well as understand the timing profile created using TAU. Figure 3 below depicts the ALF algorithm processing a 720x480 resolution image.

Figure 3. Left Original Image Vs. Right ALF Filtered Image

The parallelizable code is then ported and another timing profile is conducted to determine the increased performance. As well, some portions of serial code are also ported to the GPU and processed serially in order to reduce memory transfers between the host and device. Determining whether transferring the data or processing the data directly on the GPU requires the timing profile and a clear understanding of the GPUs performance. Thus it is important to benchmark the performance of the system before optimizing the host and device code.

3. Results and Discussion

3.1 Memory Declarations CPU and GPU

When integrating CUDA C into existing C/C++ code there are three available methods to allocate host memory: new, malloc, and cudaHostAlloc. The cudHostAlloc is similar to malloc except that it allocates a buffer of page-locked host memory. Thus, the memory will not be paged out to disk and the GPU can utilize its direct memory access (DMA) engine to copy data to and from the host. This is especially important for performance enhancement because many memory copies between the host and device will result in decreased performance. Figure 4 depicts the processing speed up achieved by using cudaHostAlloc to page lock memory for data transfers between the host and device. However, overuse of page locked memory can cause decreased system performance.

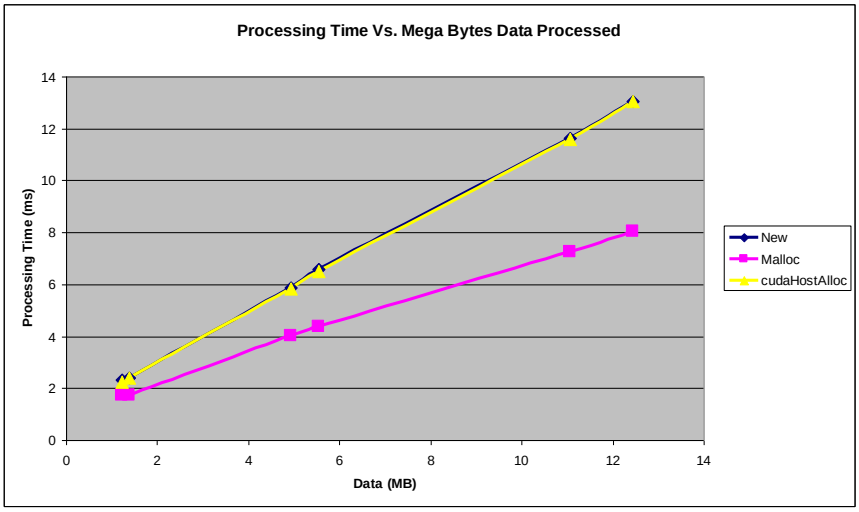

Figure 4. Data Processing Using: New Vs. Malloc Vs. cudaHostAlloc for Memory Allocation

3.2 Asynchronous Memory Transfers and Execution using Streams

NVIDIA's programming guide recommends that streams be created to properly queue device instructions to utilize the asynchronous memory copy and kernel executions. Asynchronous memory transfers and kernel executions require a GPU with concurrent copy and execution available as well as the memory being copied between host and device to be page-locked (host memory allocated using cudaHostalloc). It is important to remember that asynchronous memory transfers and computations should be utilized for portions of code which are independent of other memory transfers and computations. Figure 5 provides an example of an improperly queued stream which does not utilize the concurrent copy and execute available on the GTX 460. Highlighted in red are instructions which have been queued for stream 0 and instructions queued on stream 1 in black.

```
cudaMemcpyAsync(dataA0, stream0, HostToDevice)
cudaMemcpyAsync(dataB0, stream0, HostToDevice)
kernel<<< blocks, threads, stream0>>>(result0, dataA0, dataB0)
cudaMemcpyAsync(result0, stream0, DeviceToHost)
cudaMemcpyAsync(dataA1, stream1, HostToDevice)
cudaMemcpyAsync(dataB1, stream1, HostToDevice)
kernel<<<blocks, threads, stream1>>>(result1, dataA1, dataB1)
cudaMemcpyAsync(result1, stream1, DeviceToHost)
```

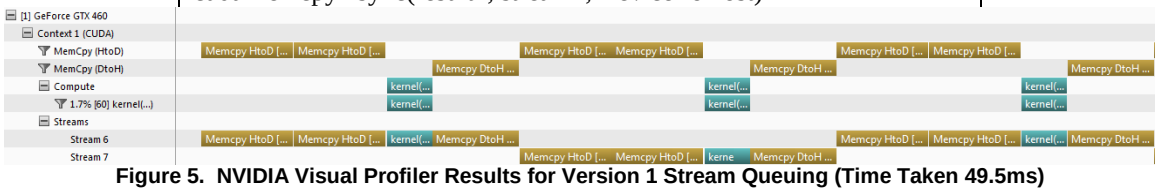

Figure 5. NVIDIA Visual Profiler Results for Version 1 Stream Queuing (Time Taken 49.5ms)

It can be seen in Figure 5 that by queuing the device instructions of stream 0 and stream 1 in a serial manner the memory copies (tan) and computations (teal) are not overlapped and no performance increase is achieved. The proper CUDA stream queue structure would be interleaving the two streams so that as memory is being copied the kernel can be instructed to execute. Figure 6 demonstrates the proper use of CUDA streams.

```
cudaMemcpyAsync(dataA0, stream0, HostToDevice)
cudaMemcpyAsync(dataA1, stream1, HostToDevice)
cudaMemcpyAsync(dataB0, stream0, HostToDevice)
cudaMemcpyAsync(dataB1, stream1, HostToDevice)
kernel<<< blocks, threads, stream0>>>(result0, dataA0, dataB0)
kernel<<<blocks, threads, stream1>>>(result1, dataA1, dataB1)
cudaMemcpyAsync(result0, stream0, DeviceToHost)
cudaMemcpyAsync(result1, stream1, DeviceToHost)
```

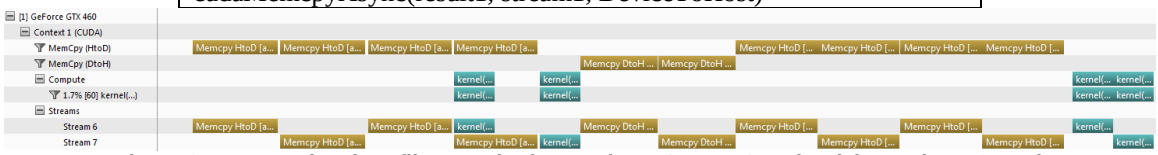

Figure 6. NVIDIA Visual Profiler Results for Version 1 Stream Queuing (Time Taken 49.4ms)

However, when using the visual profiler we see that after the first asynchronous memory copy and transfer the subsequent calls result in a serial behavior. Through testing it was found that when the kernel execution of stream 0 is immediately followed by the stream 0 memory copy, and similarly for stream 1, that full asynchronous copy and execution is achieved.

In figure 7 it can be seen that asynchronous memory copy and kernel executions overlap when queuing device instructions in this form. Thus, a portion of the memory transfer overhead is hidden during the kernel execution. For the GTX 460 only one copy engine is available. If two copy engines are available performance would increase even more as less overhead is required for memory copies.

```
cudaMemcpyAsync(dataA0, stream0, HostToDevice)
cudaMemcpyAsync(dataA1, stream1, HostToDevice)
cudaMemcpyAsync(dataB0, stream0, HostToDevice)
cudaMemcpyAsync(dataB1, stream1, HostToDevice)
kernel<<< blocks, threads, stream0>>>(result0, dataA0, dataB0)
cudaMemcpyAsync(result0, stream0, DeviceToHost)
kernel<<<blocks, threads, stream1>>>(result1, dataA1, dataB1)
cudaMemcpyAsync(result1, stream1, DeviceToHost)
```

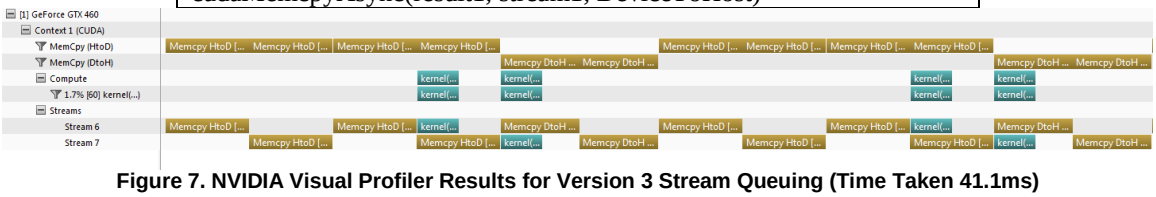

Figure 7. NVIDIA Visual Profiler Results for Version 3 Stream Queuing (Time Taken 41.1ms)

Figure 8 depicts the increase in performance when comparing proper use of streams with no streams (streams queued in serial). As more data is processed the effect of proper stream queuing becomes more evident. Thus, utilizing streams lends to scalability and performance increase in the future for larger data structures.

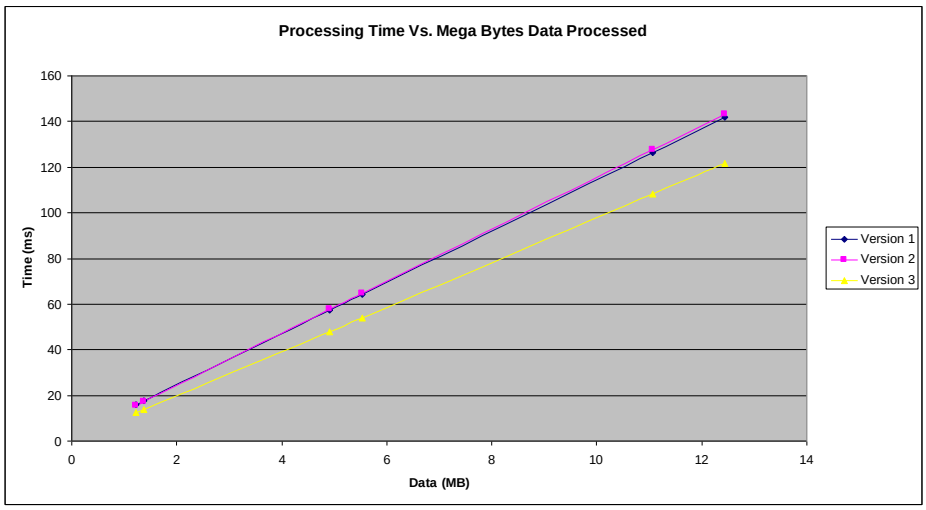

Figure 8. No Stream, Stream, and Modified Stream Queue Structure Processing Results

However, asynchronous memory copies and concurrent kernel executions utilized by proper device stream queuing does not always result in a performance increase. This is the result of the overhead associated with creating, synchronizing, and destroying streams . Thus it is important to profile the device code to ensure that a sufficient amount of processing time occurs during computations of the kernels and not during the memory copying process. Figure 9 depicts these results revealing that when computation time is low in comparison to copying time, concurrent copy and computation results in degraded performance.

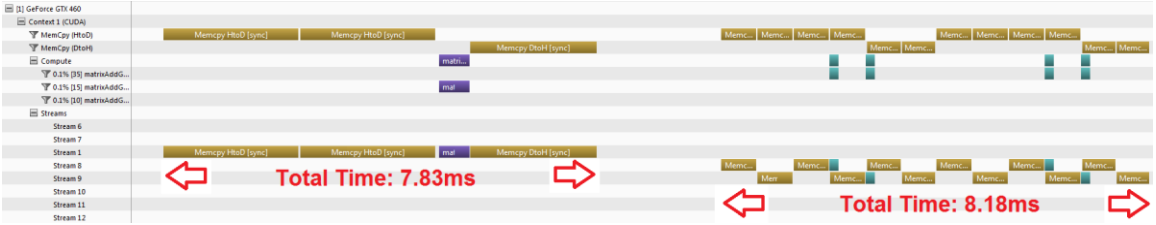

Figure 9. Visual Profile of Serial Device Copy-Compute-Copy and Concurrent Copy-Compute

3.3 GPU Processing Results

The ALF algorithm was ported to the GPU and profiled. Using the timing profile of the CPU and GPU processing times the code was then optimized by determining which data structures should be page-locked and thus able to be streamed for asynchronous memory copy and kernel execution. Table 1 presents the ALF algorithm processing time for a single frame of varying image resolutions. It can be seen that after optimization an increase in frames per second (FPS) is achieved.

Table 1. Complete ALF Algorithm Execution Time

Resolution	ALF: GPU+CPU (FPS)	ALF: GPU+CPU Optimized (FPS)	Speed Up (FPS)
720x480	7.59	17.81	10.22
1440 x 960	5.06	8.78	3.72
2160 x 1440	3.24	4.52	1.28

4. Conclusion

This paper has demonstrated that simple optimization strategies such as using page-locked host memory, asynchronous memory copies and concurrent kernel executions, and proper stream coordination results in a decrease in processing time. However, it has also been demonstrated that utilizing streams to hide the latency from device to host and host to device memory copies does not always result in increased performance. Hiding the memory copy latency works best when the computation times are comparable to memory copy times so that the stream creation, synchronization, and destruction overhead does not add more latency than it is able to hide.

Thus, developing a decomposition strategy in conjunction with benchmarked CUDA overhead requirements will help develop an optimization strategy to utilize the full computational power available through GPUs. This paper has shown that a speed up of ~2x is achieved for the ALF algorithm operating on an 720x480 resolution image.

As described previously the GPUs have their limitations, such as the large overhead associated with memory copies between host and device and vice versa. However, for image processing GPUs can be an asset. The GPUS's size-to-performance also makes it the ideal technology for a range of military applications such as UAV onboard data processing and mobile HPCs for forward operating bases.

Acknowledgements

This work has been funded by SPAWAR System Center Pacific Internal Applied Research (IAR) program (Dave Rees Program Manager). We would like to thank David Buck for allowing us to use his ALF algorithm. As well, if not for the support from our Department Manager, George McCarty, Code H Business Deputy, Neal Miyake, H5600 Division Head, Alan Umeda and H56D0 Branch Head, Justin Lee we would not have had this great opportunity. Last, and definitely not least, we would like to thank Nick Kamin and Dr. Randy Shimabukuro for mentoring us throughout the course of this applied research.

References

CUDA C Best Practices Guide Ver 4.0, 5/2011.

NVIDIA CUDA Programming Guide Ver 4.0, 5/6/2011.

Jason Sanders, Edward Kandrot. CUDA By Example, An Introduction to General-Purpose GPU Programming. Addison-Wesley. Copyright NVIDIA Corporation 2011.