Approaches to General Purpose GPU Acceleration of Digital Signal Processing in Optical Coherence Tomography Systems

K.I. Kapinchev and F.R.M. Barnes
School of Computing
University of Kent
Canterbury, UK
{kik2,F.R.M.Barnes}@kent.ac.uk

A. Bradu and A.Gh. Podoleanu
School of Physical Sciences
University of Kent
Canterbury, UK
{A.Bradu,A.G.H.Podoleanu}@kent.ac.uk

***Abstract*—This paper explores and evaluates two approaches designed to improve the performance of digital signal processing within optical coherency tomography systems. Such systems rely on high-speed cameras or fast photo detectors to capture data, that is then processed using established techniques (typically Fourier transforms) and results presented to the user. In certain applications of these systems, medical imaging in ophthalmology for instance, performance is an issue. CPU processing cannot keep pace with data capture, resulting in lost frames and longer latencies from scan to results. To address these issues, this research explores the use of commonly available graphics processors for the bulk of data processing in such systems. We have integrated this within an existing system, developed using National Instruments' LabVIEW software, and report on the results.**

***Index Terms*—parallel computing; GP-GPU; CUDA; digital signal processing; optical coherence tomography.**

I. Introduction

Optical Coherence Tomography (OCT) relies on the fact that electromagnetic radiation at certain frequencies, typically laser light, has the ability to penetrate solid materials, including tissues and organs, to some depth and reflect back information about the sample under test [1]. Unlike ultrasound imaging, where high frequency sound is used to generate imagery, OCT uses interference of light between the laser source and that reflected from the sample. OCT has two main advantages over traditional (and typically invasive) approaches: it operates with a low energy laser, safe for live specimens, and it is capable of producing high-resolution imagery. OCT has applications in a number of different areas, most notably medicine (ophthalmology and cardiology) and art (painting) investigation. In both these areas the ability to see below the surface of a sample, in a non-invasive and non-destructive manner, is critical.

Contemporary OCT systems [2] use either high-speed cameras under broadband illumination [3] or fast photo detectors under tunable lasers [4]. These are connected to a computer system via an expansion board and appropriate drivers, providing a digitised version of the input signal for data processing and visualisation. Given the physical characteristics of the system, a significant amount of signal processing (based on Fourier transforms) must be performed on the input data [5] before the results (a visual representation of the sample under test) can be obtained. For the benefit of the reader, Fig. 1 shows the physical structure of the OCT system with which we are working (typical in medical imaging).

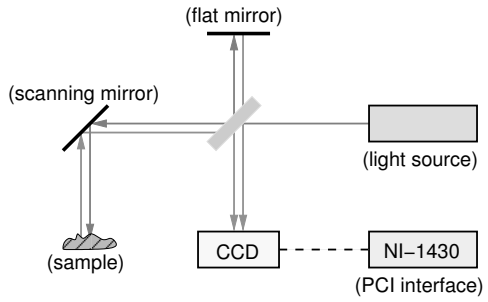

Fig. 1. Physical OCT system.

On the software side, and as a starting point for the work presented here, the OCT system shown uses National Instruments' LabVIEW for data acquisition (from the camera interface board) and processing. LabVIEW is also used to present the constructed image to the user, along with various software controls that manipulate both hardware and software components of the system.

When such systems are initially constructed, performance is not the primary concern. More important is producing systems that function correctly, both physically and mathematically. However, as more complexity is added and as data processing becomes more intensive, performance does become a concern. Moreover, a lack of performance is potentially a barrier to both future research and commercial adoption.

In high-performance computing, graphics processing units (GPUs) are increasingly being used to perform complex calculations rapidly, utilising the massive parallelism available in these devices. High-end GPUs, such as NVIDIA's TESLA [6] and GeForce GTX based graphics cards, incorporate large numbers of processor cores and well-engineered memory hierarchies designed for high-bandwidth low-latency processing.

A. Objectives and Contribution

Here we explore how a GPU can be utilised to improve the performance of an existing OCT system, developed using LabVIEW, with minimal modifications to the existing system and in a way that could be applied to other similar systems in the future. We present two relatively straightforward approaches to achieve this; straightforward in the sense that integration with the existing system can be done easily and without specialist computing or GPU knowledge.

The following section gives some further background on the technologies and techniques involved, including LabVIEW, the data processing involved, GPUs, and brief details of how the performance of such systems (in general) can be improved. Section III gives details on the approaches investigated as part of this work, with initial performance results in section IV. Initial conclusions and some discussion of future work are given in section V.

II. Background

A. LabVIEW

A typical use of LabVIEW, as exemplified here, is for constructing control systems in a visually intuitive way, using data-flow between 'VI' (virtual instrument) components. For the end-user, a suitable "front panel" interface can be constructed that provides various controls and readouts, connected to VI components as appropriate. As a result of the way the system is expressed, i.e. as *data flow* between components, and the flexibility afforded by LabVIEW, good performance can be hard to predict or obtain.

For medical imaging using OCT, including the exploration of alternative (and potentially more complex) hardware, data-flow and control configurations, something approaching or achieving real-time performance is highly desirable. More responsive and more substantial or feature rich imaging has clear benefit in medicine and elsewhere.

B. Data Processing in OCT

In OCT systems, the reflected beam of electromagnetic radiation (typically infra-red laser light) carries a large volume of information. Different layers within the sample reflect or scatter the light differently, which is captured (ultimately) as 1280 data-points of interference pattern representing the various layers/depths for a single point on the sample, determined by the scanning mirror.

To construct an image that accurately shows the sample, a large amount of signal processing is required. Much of this computational work-load is based on forward and inverse Fourier transforms as part of *cross-correlation* between two input signals. Cross-correlation, widely used in interferometry, measures the similarity and difference between two signals and can be calculated via the product of Fourier transforms of the signals involved [7], specifically:

$$X \star Y = \mathrm{F}^{-1}\big(\mathrm{F}(X) * \overline{\mathrm{F}(Y)}\big) \tag{1}$$

where X and Y are the source signals, functions F and F^{-1} are forward and inverse Fourier transforms, and $\overline{\mathrm{F}(Y)}$ is the complex conjugate of some $\mathrm{F}(Y)$. From the above, it is clear that Fourier transforms dominate the computational cost. High-performance implementations of Fourier transforms for CPUs and GPUs have been well studied [8], [9], which we make good use of here.

Given that speeding up the computation itself is not a significant issue (other than for later optimisation) the problem becomes integrating this into the existing (software) system without significant effort or re-development, and without damaging the existing data-flow oriented structure.

C. GPU Programming

To be widely applicable, this work assumes relatively little about the GPU, other than it being a parallel computing resource. Whilst recent GPUs provide a wealth of features to enhance general-purpose programming, we only require that it is capable of Fourier transforms (as provided by NVIDIA's CUDA libraries or from elsewhere) and other general arithmetic/logic operations, e.g. for signal filtering. In practice we use NVIDIA's CUDA language (essentially C with GPU-specific extensions) [10], as it is widely known, relatively stable and well tested, and compatible GPU cards are already present in the OCT system.

Although we use the GPU as a coprocessor for signal processing, in the systems considered the same card provides the primary video output. This creates some hardware contention, and depending on the particular card, can introduce significant overheads. We do not attempt to minimise or hide these overheads, but merely note their presence.

D. Improving Performance

In addition to the GPU approach considered here, there are other approaches to improving the performance of systems such as these, but at a cost — significant in hardware, software or a lack of reusability. These include:

- The use of dedicated FPGA accelerator hardware that can be integrated cleanly with LabVIEW, the significant cost primarily being hardware [11].
- Taking advantage of existing multiprocessor and multi-core hardware. To do this in a straightforward way within an existing LabVIEW system requires additional software (plug-in components designed for parallel computing). Such components also exist for utilising GPU resources, but at a cost and not necessarily with optimal performance.

Another approach is to re-develop the entire system in a language such as C or C++, utilising CPU cores and GPUs to their full potential. Whilst such systems undoubtedly have (near) optimal performance, specialised hardware aside, developing them comes at significant cost, and it may be hard for non-specialists to be able to re-use parts of the system elsewhere as they will likely be tailored to specific hardware and algorithms.

III. Proposed Approach

The proposed GPU implementation performs Fast Fourier Transforms (FFTs) and cross-correlation on signal data using CUDA library functions provided by NVIDIA, namely

"cufftExecR2C" and "cufftExecC2C". These are fairly general, catering for forward and inverse FFTs on data in up to 3 dimensions, working with both real "cufftReal" and complex "cufftComplex" types. As with user-defined CUDA kernels, the FFT functions work with data stored in the GPU's own memory [12].

The only portable and practical way to integrate external code with LabVIEW is through the use of a Dynamic Link Library (DLL). A specific component in LabVIEW can be configured to load and call functions inside a DLL, passing pointers to data buffers [13]. The execution of the DLL call with respect to the rest of the LabVIEW system is not clearly documented. In the standard (non-augmented) version of LabVIEW used, the observed behaviour is that the system is essentially suspected whilst the DLL (function) call takes place.

Two approaches are investigated. First, building the CUDA code into a DLL which is then called from LabVIEW, and second, building a standalone CUDA application that exchanges data with LabVIEW via a DLL and shared-memory. In both cases, the called function has the same structure:

```
void do_oct_dsp (float *in, long size,
                 float *out)
{
    // CUDA routines
}
```

A. CUDA Functions in a DLL

An obvious approach to integrating CUDA code is to simply compile it into a DLL using appropriate tools — NVIDIA's CUDA compiler and Microsoft's Visual C/C++ compiler (part of Visual Studio) in this case. This is shown in Fig. 2.

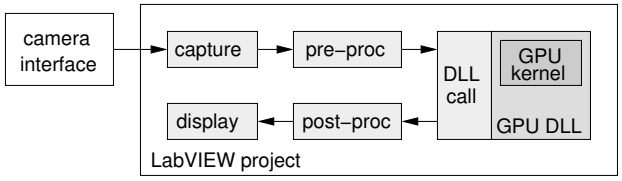

Fig. 2. DLL implementation of CUDA DSP.

Although this is straightforward, there is a significant issue in that the GPU context is re-created each time the function is called. This includes allocating and releasing memory on the GPU, in addition to copying data to/from main memory and launching the kernel. For the array sizes involved and the frequency with which the function is called, this approach does not yield good performance.

B. Standalone CUDA Application

In the second approach, shown in Fig. 3, the GPU code is compiled into a standalone application, that exchanges data with LabVIEW using a DLL and shared-memory. This ensures that the GPU context is created only, but is less transparent to the end-user, who must launch this application separately to the LabVIEW system.

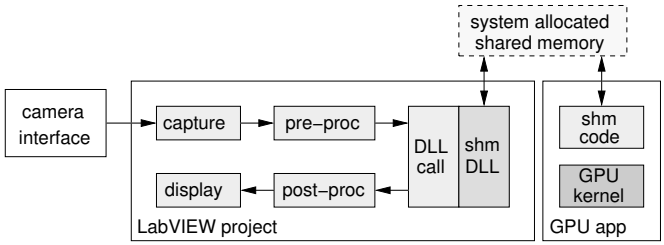

Fig. 3. Independent implementation of CUDA DSP.

The DLL to which LabVIEW interfaces is responsible for reading and writing data to a shared-memory region (treated as a *page-mapped file*) and notifying the CUDA application. The CUDA application then copies the data to the GPU (into a pre-allocated region) and launches the kernel. On completion, the results are copied back into the shared memory area and the waiting DLL notified. The overall operation of this is shown in Fig. 4.

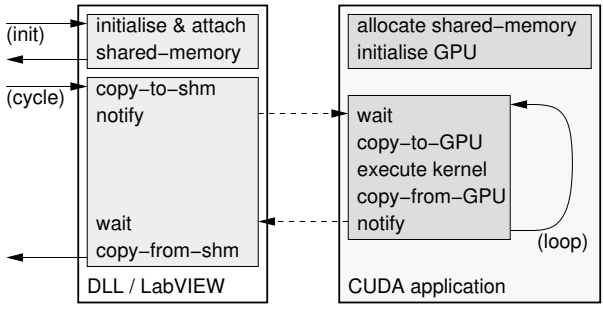

Fig. 4. Interfacing LabVIEW with CUDA via shared-memory.

Two different implementations of the 'wait' and 'notify' are used. The first uses Windows *semaphores* to suspend execution of the relevant process until the shared memory area has been populated. The second uses *busy-waits* on part of the shared area, interleaved with 'yield' system calls.

IV. Performance

Generating accurate performance timing measurements on a Microsoft Windows platform is not straightforward in general. Whilst millisecond timers are standardised (in the Windows API) much of what we wish to measure is sub-millisecond. Unix based systems such as Linux feature nanosecond timers [14] that make benchmarking a somewhat easier prospect. The common approach to sub-millisecond timing on Windows is to use the hardware cycle counter where present (available on most x86/IA32/IA64 based systems), accessed by two system calls: one to read the current value and one to report the number of cycles per second. A slight caveat is that any dynamic CPU frequency scaling (usually for power-saving and/or brief over-clocking) must be disabled.

From the various approaches described previously, the best performance is obtained when using a standalone application (meaning the GPU context is only created once) communicating with LabVIEW using shared-memory, and synchronising

using busy-waits. Fig 5 shows the times for various GPU operations, measured from within the standalone application for various sizes of data (in arrays of 32-bit floating-point values), average over a number of runs. The GPU used is an NVIDIA GeForce GT 630, with 96 CUDA cores, that is also used for display output.

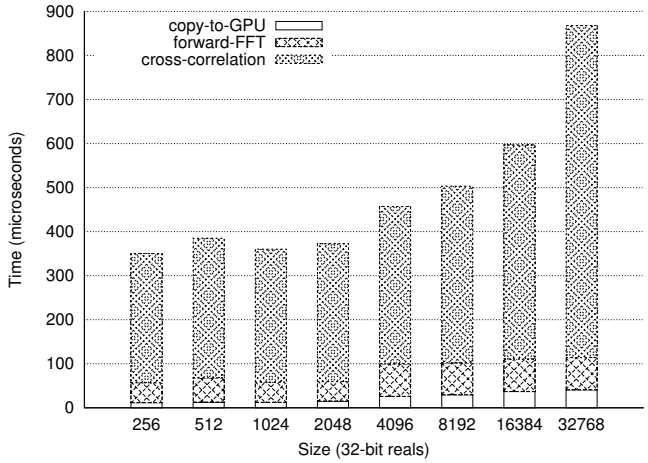

Fig. 5. Benchmark results for GPU operations.

The default operation of the CUDA libraries permit some latency hiding (by performing asynchronous memory copies). These results are representative of how the application behaves in real-time, running alongside the LabVIEW application. As can be seen, the processing time for cross-correlation dominates. To get a better understanding of the overheads involved, Fig 6 shows results for the same benchmarks, but with the GPU device synchronised after each operation.

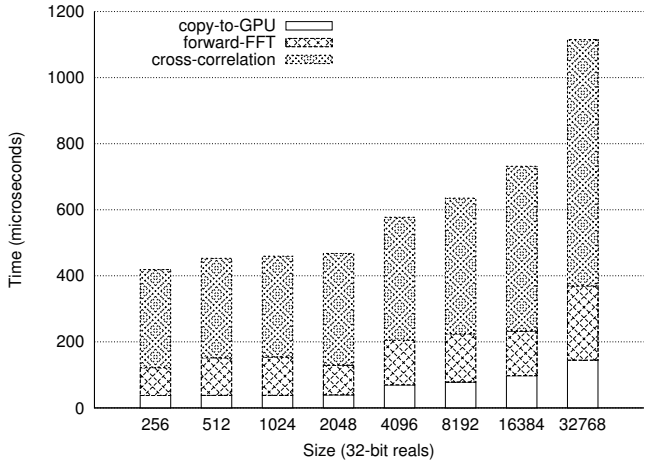

Fig. 6. Benchmark results for GPU operations with stream synchronisation.

The results show that the latency hiding mechanisms employed by CUDA do provide a significant benefit, but can cause benchmark results to appear slightly distorted. When using OS semaphores to control interactions with LabVIEW, performance of the whole system is significantly reduced, although there is no significant change in the individual GPU overheads.

V. Conclusions and Future Work

The nature of the data processing within OCT makes it suitable for parallel execution using commonly available GPUs. Here we have shown how this may be integrated into an existing LabVIEW system, without extensive modifications to that system and with limited additional cost (save for development time). The approach that provides the best performance uses a standalone application that runs continuously, exchanging data and synchronising with a running LabVIEW system via shared-memory.

As can be seen from the results presented, the cross-correlation operation dominates. Unlike the FFT library implementation used, we have not optimised this code — compared with the FFT overheads, this suggests that better performance could be obtained with some optimisation effort.

To discover what level of performance is possible within the hardware constraints, we plan to develop a C/CUDA application that implements (effectively) the entire OCT system, including input from the camera and control of other system components. Such an approach has the advantages of compiled code and fewer memory copies, as well as direct-to-display video output via OpenGL, but at the expense of some generality and flexibility. We hope that it will be possible to re-use some parts of such a system however, e.g. for direct visualisation output, without needing to transfer results from the GPU back into host memory. Moreover, we hope to be able to integrate such components with existing LabVIEW systems with minimum effort in order to gain better performance using hardware already in place.

Acknowledgements

The authors acknowledge the support of the University of Kent, the EU (FP7 grant 249889), the NIHR Biomedical Research Centre, and the UCL Institute of Ophthalmology.

References

[1] W. Drexler and J. G. Fujimoto, *Optical Coherence Tomography: Technology and Applications*. Springer, 2008.

[2] A. G. Podoleanu, "Optical coherence tomography," *Journal of Microscopy*, vol. 247, no. 3, pp. 209–219, 2012.

[3] A. Bradu and A. G. Podoleanu, "Fourier domain optical coherence tomography system with balance detection," *Optics Express*, vol. 20, no. 16, pp. 17 522–17 538, 2012.

[4] T. Klein, W. Wieser, C. M. Eigenwillig, B. R. Biedermann, and R. Huber, "Megahertz OCT for ultrawide-field retinal imaging with a 1050nm fourier domain mode-locked laser," *Optics Express*, vol. 19, no. 4, pp. 3044–3062, Feb. 2011.

[5] S. Van der Jeught, A. Bradu, and A. G. Podoleanu, "Real-time resampling in fourier domain optical coherence tomography using a graphics processing unit," *Journal of Biomedical Optics*, vol. 15, no. 3, p. 030511, 2010.

[6] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Kepler GK110," 2012.

[7] D. Lyon, "The discrete fourier transform, part 6: Cross-correlation," *Journal of Object Technology*, vol. 9, no. 2, pp. 17–22, 2010.

[8] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[9] S. W. Smith, *Digital Signal Processing: A Practical Guide for Engineers and Scientists*. Newnes, 2003.

[10] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[11] O. Storaasli and D. Strenski, "Exploring accelerating science applications with FPGAs," *Proc. of the Reconfigurable Systems Summer Institute*, 2007.

[12] NVIDIA Corporation, "Cuda fast fourier transform," 2012. [Online]. Available: http://docs.nvidia.com/cuda/cufft/index.html

[13] National Instruments Corporation, "Writing win32 dynamic link libraries (DLLs) and calling them from LabVIEW," 2010, white paper 4877. [Online]. Available: http://www.ni.com/white-paper/4877/en/

[14] IEEE and The Open Group, "IEEE Std 1003.1, 2004 edition (derived from POSIX.1:2001,Single Unix Specification)," 2004. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/