Accelerating a MATLAB Application with Nvidia GPUs: a Case Study for GPU Library Construction

Nicholas Moore and Miriam Leeser
{nmoore, mel}@coe.neu.edu
Dept. of Electrical and Computer Engineering
Northeastern University, Boston, MA

This abstract covers a case study in using the Nvidia CUDA environment to accelerate lung tumor tracking through template matching, and will discuss how this lays the groundwork for creating high performance GPU accelerated applications through a modular library-based programming interface. The tracking algorithm uses motion-enhanced templates and Pearson's correlation. Issues that arise when using CUDA from within MATLAB are investigated, as well as how the particular application was mapped from MATLAB code to a GPU. The case study is being used to develop a tool that allows more rapid exploration of the tradespaces for GPU applications.

The goal is to develop techniques that support abstracting user code from the architecture specific code and to provide portability along with high performance.

Template Matching Application

Radiotherapy for lung tumors is commonly handled by the administration of radiation over a large area since lung tumors can move significantly during normal breathing. This radiates normal tissue as well, necessitating the use of a low-intensity beam. Y. Cui, *et al.*[1] are researching ways to improve lung tumor radiotherapy through the use of focused higher-intensity radiation that tracks the tumor. The tumor tracking algorithm considered in this case study generates motion-enhanced templates and uses Pearson's correlation for its similarity function. The original templates, generated by a clinician using a tool developed with MATLAB to mark the tumor location at various representative locations through the patient's respiration cycle, are converted to motion-enhanced templates by subtracting a given template from the average of all the templates, thus emphasizing moving portions of the image.

$$corr2(A,B)=\frac{\sum_M\sum_N (A_{MN}-\bar{A})(B_{MN}-\bar{B})}{\sqrt{(\sum_M\sum_N (A_{MN}-\bar{A})^2)(\sum_M\sum_N (B_{MN}-\bar{B})^2)}}$$

Figure 1: Pearon's Correlation

To handle further variation in respiration, the Pearson's correlation similarity function, shown in Figure 1, is applied to multiple locations, forming a region of interest (ROI), as shown in Figure 2. The similarity function is computed for each ROI location for each template and for each image to determine the likeliest location of the tumor.

The result of computing so many individual similarity scores is a significant amount of processing. For the six patient data sets used for algorithm development, the algorithm requires from about 279,000 to about 3.73 million correlations, taking from about 77 seconds to 592 seconds to execute (on an Intel Core 2 Duo E8400) and a relatively constant 82% of the total application runtime across all six patients.

The significant time spent processing images makes this application a good match for GPU architectures. To get maximum parallelism out of the GPU, all of the Pearson's correlation similarity functions, implemented by the *corr2()* function in MATLAB, are executed together in parallel.

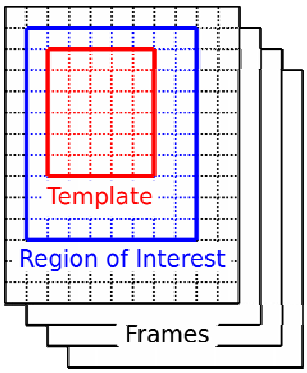

Figure 2: Multiple 2D Correlations

Working with CUDA

The current Nvidia GPU architecture offers a lot of potential processing power, but requires developers to structure their application around it. The CUDA environment provides programmers with more complete control over the placement of data in memory than is common with CPU architectures. The various types of memory, each with different performance characteristics and sizes, are characterized for the GPUs considered in this case study in Table 1. Processing elements are grouped into multiprocessors that share cached read-only constant and texture memories and read/write shared memories. The register pool is also allocated at the multiprocessor level. Device memory is the largest and has the fewest restrictions, but can be much slower than constant and texture memory. Shared memory allows subsets of the individual GPU processing elements to communicate relatively efficiently during kernel execution. A given GPU kernel's output must always be placed in device memory.

In addition to performance characteristics, the host-based control code to set up a given kernel's inputs also varies between memory types. Texture memories require the setting of a number of parameters, and binding data to texture or constant memory may require a data copy between locations within a GPU's device memory.

GPUs are also different from CPUs in that individual processing elements are less capable than a general purpose CPU core. With the GPUs considered in this case study, compute-bound kernels generally want to use ten or fewer registers (for architectural reasons not obvious from Table

1). Spilling out of the registers allocated to an individual processing unit is often prohibitive. However, the register count may be increased by careful utilization of shared memory as extra register space.

Type	Size	Scope	R/W	Speed
Registers	32 KB	Thread	RW	Fast
Shared	16 KB	Multiprocessor	RW	Fast
Device	1.5 GB	Global	RW	Slow
Constant	8 KB cached	Global	R	Fast when cached
Texture	6-8 KB cached	Global	R	Fast when cached

Table 1: Nvidia GPU Architecture Memories per Multiprocessor

In response to these constraints, the parallel *corr2()* function considered here, for which example parameter sizes are presented in Table 2, was broken into six steps: ROI and template averages, ROI and template standard deviations, the numerator, and final multiplication.

Template Dimensions	21x23 pixels
Vertical Shift	±11 pixels
Horizontal Shift	±5 pixels
Templates	13
Frames	348
corr2() Operations	1 144 572

Table 2: Parameter Information for a Particular Patient

This allows the implementation to take advantage of the large amount of computational redundancy across the set of *corr2()* executions for a given patient: both the matrix averages and the sums of the squares of the deviations from the matrix means are computed once and reused.

To explore the performance trade-offs associated with each type of memory, different memory locations for the kernel inputs have been implemented. Different GPU kernels are required for each combination of memory locations for inputs, as the GPU code for accessing each is slightly different. As an example of a trade-off, it is not known whether the caching provided by texture memories will provide enough of a performance benefit for a given step to outweigh the extra setup required over global memory.

Decisions about how to map an application to CUDA must take these and many other considerations into account, resulting in a large tradespace.

MATLAB Interfacing

With this application, consideration was given to interfacing with MATLAB, which was performed through a MEX file. This involves both data type conversion and memory allocation issues. Only more recent Nvidia GPUs support double precision arithmetic, and those that do suffer about an order of magnitude hit in performance relative to single precision arithmetic. As a result, it is still desirable to execute GPU computations in single precision. Since MATLAB's default is double precision, it is necessary to convert the data, adding overhead to the application.

In addition to conversion issues, the CUDA environment recognizes multiple types of host memory with different host-to-device data transfer characteristics. A data copy from MATLAB allocated memory to optimal CUDA allocated host memory can be absorbed into the data conversion process. The data transfer choices also add to the size of the tradespace.

Since the input data will already be streaming through the CPU for conversion, executing some of the application steps with the CPU instead of the GPU is being examined. This can reduce the number of kernel launches, especially for inefficient kernels, amortizing conversion overhead. For example, the template average calculation is over a small number of templates and doesn't result in enough parallelism to fully utilize a GPU. Another option is to force the entire MATLAB/GPU computation to take place in single precision, which may combine with the other execution options resulting in trade-offs related to performance, memory type usage, and computational accuracy.

Conclusions & Future Work

A good understanding of a GPU's memory hierarchy is important since memory hierarchy is manually managed and essential to achieving high performance on GPUs. The case study presented here, which will be completed with results by the time of the HPEC Workshop, is being used to build a tool that enables quicker exploration of the implementation tradespace by expediting the creation of various instances. It is the first step towards a library-based programming interface for GPUs. Knowledge of the hierarchy not only allows efficient implementation of GPU kernels, but also sheds light onto what aspects of the kernels will need to be characterized and parameterized for a GPU library to effectively map application descriptions onto the execution hardware.

Future work includes examining alternate mappings of the *corr2()* function's mathematics to GPU kernels. In addition to the particular setup used for this case study, the trade-offs will also be examined on various CUDA-enabled GPUs as well as double precision capable GPUs. Studying other vendors' products, such as AMD/ATI, and other environments, like OpenCL, will be examined.

Acknowledgements

This research was supported, in part, by the The MathWorks.

References

[1] Y. Cui, J. G. Dy, G. C. Sharp, B. Alexander, and S. B. Jiang, "Multiple Template Based Fluoroscopic Tracking of Lung Tumor Mass without Implanted Fiducial Markers," *Physics in Medicine and Biology*, Vol. 52, pp. 6229-6242, 2007.