Experience Porting MATLAB Systems Biology Applications to CUDA

LAVA: Laboratory for Computer Architecture at Virginia
University of Virginia, Charlottesville, VA 22904

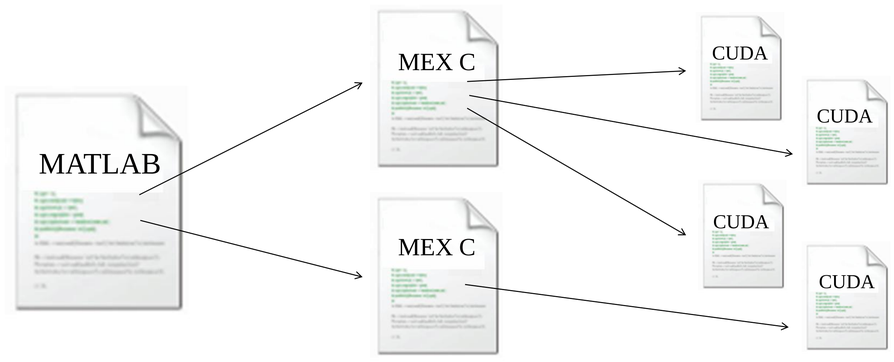
Lukasz Szafaryn, Michael Boyer, Kevin Skadron

http://lava.cs.virginia.edu

This work is supported by a grant from NVIDIA Research and NSF grant no. CNS-0615277.

Introduction

Systems biology seeks to develop an understanding of the myriad interacting components of biological systems. Various modes of biomedical imaging provide a rich source of data for building and testing models. However, accurate modeling requires massive parameterization, which in turn requires image extraction, tracking, and mining for relationships. GPU computing and CUDA offer the potential for substantial speedups. However, direct porting to CUDA is not always possible due to the preference of many systems biologists to use MATLAB. We are investigating how to best obtain the benefits of GPU computing while allowing the biologist to continue using MATLAB. There are usually two conflicting goals involved in accelerating MATLAB applications with CUDA: convenience of porting and performance.



Structure of MATLAB application accelerated with CUDA: interpreted MATLAB code (a) offloads tasks to compiled C code (b) for faster execution, which is further accelerated by offloading parts of work to parallel GPU kernels (c).

Tradeoffs

Developing modular code
- replacing each MATLAB function with equivalent parallelized GPU function
- common routines can be possibly reused in other applications

Hiding specific aspects of GPU programming inside each module
- each module sets up its own GPU execution parameters
- each module performs its own I/O with GPU transparently

Convenience

Performance

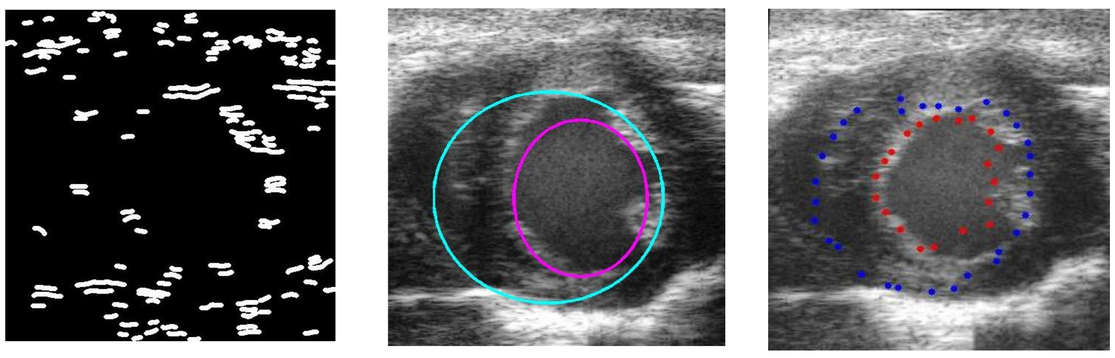
Restructuring and combining code
- writing code based on algorithm tasks rather than MATLAB statements
- overlap parallel (often unrelated) tasks by executing them in the same kernel call

Exposing specific aspects of GPU programming
- doing more work inside each GPU kernel call to fully exploit parallelism and avoid overhead
- performing I/O with GPU manually to eliminate redundant data transfers

Heart Wall Tracking

An Example Application

A particularly challenging application we are working with tracks the movement of the inner and outer walls of a mouse heart over a sequence of 100 640x480 ultrasound images. First, the program performs several image processing operations on the first image to detect initial, partial shapes of heart walls. In order to reconstruct approximated full shapes, the program generates ellipses that best match the partially detected shapes. Ellipses are then superimposed over the image and sampled to mark points on the heart walls. Finally, the program tracks movement of the heart walls by detecting the movement of image areas under sample points as the shapes of the heart walls change throughout the sequence of images.



Stages in Heart Wall Tracking application: heart wall shape detection (a), ellipse matching (b), heart wall shape tracking (c).

Performance Results

The Heart Wall Tracking application illustrates tradeoffs in offloading computation to the GPU. Clearly, data parallelism and sufficient work per kernel are pre-requisites. However, even when the underlying algorithm is not limited by Amdahl's Law, extracting the parallelism from MATLAB may require restructuring and sacrifices in the modularity of the offloaded CUDA computation.

Major Application Part	Parallelizable Part of Code [%]	Original MATLAB Run Time [s]	Convenience Porting, Run Time [s] and Speedup [x]	Performance Porting, Run Time [s] and Speedup [x]
SRAD	89	8.71	1.26 / 6.92	1.17/ 7.42
Hough Search	87	15.87	2.97 / 5.34	2.57 / 6.17
Tracking	37	129.28	115.43 / 1.12	89.78 / 1.44
All	41	187.39	160.16 / 1.17	125.77 / 1.49

Performance numbers were obtained by running application on NVIDIA GeForce 8800GTX.

Future Work

Interesting directions for future work include automated compiler analysis within the MATLAB runtime to perform the necessary restructuring transparently, automatic analysis of whether to offload a CUDA kernel or run it locally on a CPU, and techniques to cope with tightly coupled serial-parallel steps while preserving the overall MATLAB programming "look and feel".