Parallel Programming Survey

A comparisson between multiple paralell programming frameworks

Cameron Stuart
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, United States
stuartc@oregonstate.edu

***Abstract*—This paper is a survey of using Cuda, OpenCL, and PyCuda for parallel programming with performance comparisons to a sequential implementation. A comparison between the user experience of each framework is done as well on a Windows system.**

***Keywords—CUDA, OpenCL, PyCuda, Parallelism*.**

I. Introduction

There are a variety of different frameworks available for creating parallel programming implementations that make use of a graphics card. Specifically, Cuda and OpenCL are two of the most commonly used frameworks and are compared with PyCuda, a python wrapper for Cuda, in this paper. Each one has a unique interface with varying degrees of ease for a developer to get started with. Another important consideration is the ease of setting up a development environment for each framework as each one has a variety of different steps. This survey will be comparing the development experience using each framework as well as setting up the development environments.

II. Development Environments

The first step towards developing in any of these frameworks is setting up the development environment, which is a task with varying difficulty for each language. Out of the three options, CUDA was by far the simplest to setup on a Windows development environment and consisted of running an installer put out by Nvidia. In addition to be easy to set up, the CUDA tool kit put out by Nvidia was also a dependency for the other frameworks as the toolkit contained a variety of OpenCL compilers. CUDA also contained project solutions for Visual Studio, making the entire process a quick setup to get working.

OpenCL was slightly more difficult to setup as there wasn't a unified installer released in the same was as CUDA. When using a Visual Studio development environment most of the setup involved linking OpenCL libraries and compilers. This was done mostly with the project properties and setting up system path variables, but again all those dependencies were contained within the previously discussed CUDA development kit.

PyCuda was the most challenging to setup, as configuring python packages on Windows can be more involved than a simple installer. For this survey the dependencies for PyCuda were installed by using Anaconda, a data science platform for python. Afterwards Anaconda was used to install a variety of other packages including numpy that are all dependencies for PyCuda. Afterwards the PyCuda repository on GitHub was cloned and installed using Anaconda. It should be noted that much of this process was done with trial and error as the installation steps can vary greatly upon the specific guide. This could also be a particularly painful process if the developer is inexperienced in setting up new python packages on Windows.

III. Coding and Performance

The CUDA and OpenCL frameworks are both built upon C, so developing with those is creating a C program while using their APIs. The CUDA API is relatively user friendly and abstracts away a good amount of details so setting up block and thread size is built in when launching the kernel functions. Many of the built-in functions are intuitive within the context of normal sequential C programming, while OpenCL has a much more involved API. OpenCL is similar to OpenGL in the sense that it utilizes fixed functions which rely upon a large amount of inputs, many of which are constants defined in a header file or simply null.

The performance of CUDA and OpenCL both had significant improvements over the sequential implementation. The time range for the sequential implementation was around 4ms while both parallel implementations were in the range of .1ms for the same amount of data. Between the two, OpenCL had better kernel performance taking about a tenth of the time of the CUDA kernel. That performance increase for this specific example was offset a bit about the added effort in writing the OpenCL program.

The PyCuda framework is different than the other two due to being a python wrapper. As such the interface allows for the expression of a relatively complex program with much less code and a higher level of abstraction. Additionally, PyCuda uses the same kernel code as CUDA so kernels written for either one can be used interchangeably. While running a PyCuda script takes longer than a CUDA program since it compiles the kernel at runtime, while CUDA builds it in advance, the actual kernel performance was slightly better for PyCuda compared to CUDA but still less than OpenCL.

IV. Conclusions

After implementing the same algorithm in each framework it was conclusive that there were very significant performance benefits compared to using a sequential implementation. This is dependent on the algorithm however and it shouldn't be assumed that these frameworks will always improve performance. In a Windows development environment with Visual Studio CUDA was clearly the easiest to setup and start developing in, while the various dependencies for PyCuda took much longer to initialize. OpenCL had the best performance but is a much more involved API to develop in compared to the others. Out of the three PyCuda seemed to be the best to develop in in terms of code complexity using Python compared to C.