Optimizing CUDA Parameters with a GA

Mark Harmer
mharmer@cse.unr.edu

***Abstract*—Compilers and runtimes contain many parameters that can be optimized to create a better performing program. Generally, these parameters are tuned by hand to speed up program execution. Because of the large optimization-space to search, a genetic algorithm is proposed to be used in conjunction with the CUDA compiler and runtime to find solutions for parameter sets.**

I. Introduction

Optimizing CUDA applications through the parameters used both at compile-time and run-time requires searching a large search space. There exist several static analysis methods for determining how much faster a program will run after optimizations are applied, relative to the original program [**?**]. Although static analysis is considered faster to compute, it requires increasingly complex software to model evolving hardware.

Previous work on searching for compiler optimizations on the CPU includes the exploration of parameters using a GA on the GNU compiler collection [**?**]. Additionally, the optimization-space has been explored using heuristics [**?**]. A variety of other techniques have also been used. Work done on searching the CUDA parameter optimization-space has been limited to factors such as loop unrolling [**?**].

GPU occupancy is a starting point towards discovering good optimizations. Occupancy is defined as the ratio of the number of active warps to the number of supported warps per streaming multi-processor (SM). If occupancy is at maximum, the maximum supported number of warps on an SM are being utilized. Occupancy of a SM is based on the resource usage of the following within a SM:

- Shared memory
- Thread count
- Thread blocks (currently 8)
- Register usage
- Number of warps

These factors can determine the occupancy of the GPU. Generally, a higher occupancy leads to faster results.

II. Profiling the GPU

The CUDA run-time contains a profiler that can be enabled at run-time via the environment variable `CUDA_PROFILE`. The profile also makes use of a configuration file to determine which event counters to enable. Event counters are created only for 1 SM. Event counters include the occupancy information, memory transfer sizes, number of branch instructions, etc. [**?**].

The profiler will run when the program is started and will dump a file to disk once the program is completed. This output file will contain the resulting counter values.

III. Optimizing CUDA Applications Using a GA

A method is proposed to optimize CUDA application parameters using a genetic algorithm (GA). GAs are known for finding solutions in a large search space. The GA would run on a supplied application with an initial population and future generations of solutions. Since the GA itself is not computationally expensive it will be run on the CPU and drive the simulation over each member of the population.

This method will be designed to not interfer with the source code or other compile-time complexities. This allows for a much lower learning curve for the end-user on using the application to optimize their application.

The parameters that are available for optimization are the kernel launch parameters: block size, thread size and shared memory size. Aside from register usage, the available parameters for optimizing cover the GPU occupancy concern. Because of the black-box nature of the application, a GA is suited to provide a member of the population as input to the parameters and receive the results through the profiler output file. The caveat to this system is that only parameters that are passed to the CUDA run-time can be optimized. Compiler parameters such as `maxrregcount` and loop unroll factors used in `#pragma` are excluded from optimization. This is a known limitation, considering that search space of kernel parameters may be relatively small this design will probably be changed to incorporate the other optimization paramters available at compile-time.

IV. Details on Interacting With the CUDA Run-Time

It is proposed to create a shared library under linux that will be preloaded via the `LD_PRELOAD` environment variable into the targetted process for optimization. Since `LD_PRELOAD` forces the dynamic linker to load its exported functions first, we can setup our own CUDA kernel configure and invocation functions that will be called instead of the original functions. These new functions can then be told to use different kernel configuration parameters, setting up the entry point for the GA to push a population individual's information to run the simulation. Since the shared library must be specified for the target application, a driver program is needed to accept the target application name, create the GA, and run the target application with the `CUDA_PROFILE` environment variable. The targetted application will be run several thousand times across several generations using this approach.

V. Conclusion

A proposed system for using a GA to optimize certain parameters within a CUDA application was presented. Optimal

parameter values may change across different hardware. This method for dynamic analysis of any CUDA application is considered to be generalized and flexible for future applications and hardware. The foreseeable drawback to such a system is if the CUDA run-time API is altered. The design will also attempt to incorporate compile-time parameters after the results of the run-time parameters are obtained.

Approaches can be looked at for compile-time optimizations that don't require code modifications, such as optimizing compiler arguments `maxrregcount`. Code modifications with loop unrolling can also be researched, although the user knowledge requirement of the optimization program will increase.