Efficient Matching of GPU Kernel Subgraphs

Robert Lim
Computer and Information Science
University of Oregon
Eugene, OR, USA
roblim1@cs.uoregon.edu

***Abstract*—Accelerator architectures specialize in executing SIMD (single instruction, multiple data) in lockstep. Because the majority of CUDA applications are parallelized loops, control flow information can provide an in-depth characterization of a kernel. Our methodology statically separates CUDA binaries into basic block regions and dynamically measures instruction and basic block frequencies. Our approach captures this information in a control flow graph (CFG) and performs subgraph matching across various kernel's CFGs to gain insights into an application's resource requirements, based on the shape and traversal of the graph, instruction operations executed and registers allocated, among other information. The utility of our approach is demonstrated with SHOC and Rodinia application case studies on a variety of GPU architectures, revealing novel thread divergence characteristics that facilitate end users, autotuners, and compilers in generating high performing code.**

I. Introduction

Structured programming consists of base constructs that represent how programs are written. When optimizing programs, compilers typically operate on the intermediate representation (IR) of a control flow graph (CFG), which is derived from program source code analysis and represents basic blocks of instructions (nodes) and control flow paths (edges) in the graph. Thus, the overall program structure is captured in the CFG and the IR abstracts machine-specific intrinsics that the compiler ultimately translates to machine code. In particular, compilers can benefit from prior knowledge of optimizations that may be effective for specific CFG structures.

This work presents a scalable toolkit for heterogeneous computing applications. Specifically, our approach provides a new methodology for characterizing CUDA kernels using control flow graphs and instruction operations executed. It performs novel kernel subgraph matching to gain insights into an application's resource requirements. To the knowledge of the authors, this work is a first attempt at employing subgraph matching for revealing thread divergence behavior and generating efficient code.

II. Background

A CFG is constructed for each GPU kernel computation and can be represented as a directed graph $G = (N, E, s)$, where (N, E) is a finite directed graph, and a path exists from the $START$ node $s \in N$ to every other node. A unique $STOP$ node is also assumed in the CFG. A node in the graph represents a basic block (a straight line of code without jumps or jump targets), whereas directed edges represent jumps in the control flow.

$$\begin{array}{c} \begin{array}{cccccc} R_1 & L_1 & L_4 & L_3 & L_2 & L_5 \end{array} \\ \left[\begin{array}{cccccc} .21 & - & - & - & - & - \\ 0 & .04 & - & - & - & - \\ 0 & .04 & .38 & - & - & - \\ 0 & 0 & 0 & .08 & - & - \\ 0 & 0 & 0 & 0 & .21 & - \\ 0 & 0 & 0 & 0 & .02 & 0 \end{array}\right] \end{array} \quad \begin{array}{c} \begin{array}{cccc} R_1 & L_3 & L_2 & L_1 \end{array} \\ \left[\begin{array}{cccc} .30 & - & - & - \\ 0 & .51 & - & - \\ 0 & 0 & 0 & - \\ 0 & 0 & 0 & .21 \end{array}\right] \end{array}$$

Fig. 1. Transition probability matrices for Pathfinder (`dynproc_kernel`) application, comparing Kepler (left) and Pascal (right) versions.

Transition probabilities represent frequencies of an edge to a vertex, or branches to code regions, which describes the application in a way that gets misconstrued in a flat profile. Figure 1 displays transition probability matrices for a kernel in the Pathfinder application. A stochastic matrix could also facilitate in eliminating dead code, where states with 0 transition probabilities represent node regions that will never be visited. Kernels employing structures like loops and control flow increase the complexity analysis, and transition probabilities of kernels could help during code generation.

III. Methodology

Transition probability matrices are calculated for each kernel subgraph. The entries of the transition probability matrix were calculated by dividing the number of observed node transitions i to j by the sum of $code(M)$, shown as the lower triangular elements, for some $m_{i,j}$. Note that entries of the upper triangular were discarded for convenience since the lower triangular entries represent the node transitions. Although the matrices differ in size, observe that a majority of the transitions take place in the upper-left triangle, with a few transitions in the bottom-right, for all matrices. The task is to match graphs of arbitrary sizes based on its transition probability matrix and instruction operations executed, among other information.

We statically collect instruction mixes and source code locations from generated code and map the instruction mixes to the source locator activity as the program is being run. The static analysis of CUDA binaries produces an objdump file, which provides assembly information, including instructions, program counter offsets, and line information. We attribute the static analysis from the objdump file to the profiles collected from the source code activity to provide runtime characteri-

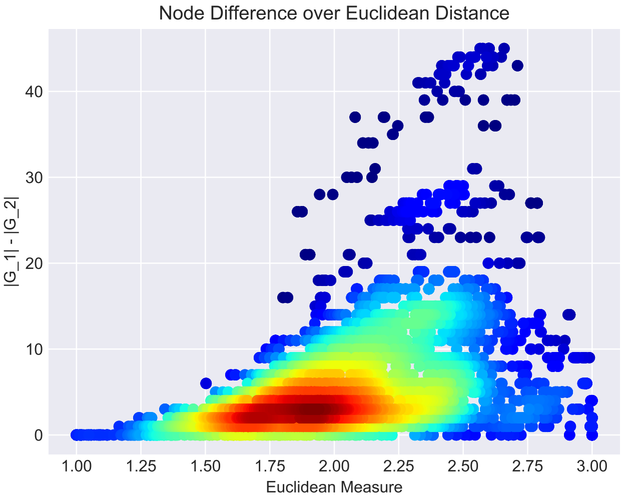

Fig. 2. Differences in vertices between two graphs, as a function of Euclidean metric for all GPU kernel combinations. Color represents intensity.

zation of the GPU as it is being executed on the architecture. This mapping of static and dynamic profiles provides a rich understanding of the behavior of the kernel application with respect to the underlying architecture.

A. Bilinear Interpolation

To scale the transition matrix before performing the pairwise comparison, we employ a spline interpolation procedure. Spline interpolation is general form of linear interpolation for functions of n-order polynomial, such as bilinear and cubic. For instance, a spline on a two-order polynomial performs bilinear interpolation on a rectilinear 2D grid (e.g. x and y). The idea is to perform linear interpolation in both the vertical and horizontal directions. Interpolation works by using known data to estimate values at unknown points.

B. Pairwise Comparison

Once the matrix is interpolated, affinity scores for the CFGs (S_1 and S_2 for graphs G_1' and G_2', respectively) are matched via a similarity measure, which includes the IsoRank solution and the Euclidean distance. By definition, $\text{sim}(G_i, G_j) = 1$ when $i = j$, with the similarity measure placing progressively greater weights on objects that are further apart.

IV. Results

Figure 2 projects the differences in vertices $|V|$ for all 162 CFG kernel pairs as a function of the Euclidean measure (application, architecture, kernel), with shade representing the frequency of the score. Note that most matched CFGs had a similarity score of 1.5 to 2.2 and had size differences under 10 vertices. Figure 2 also shows that as the differences in vertices increase, similarity matching becomes degraded due to the loss of quality of information when interpolating missing information, which is expected. Another observation is that strong similarity is exhibited when node differences of the matched kernel pairs were at a minimum, between 0 and 8.

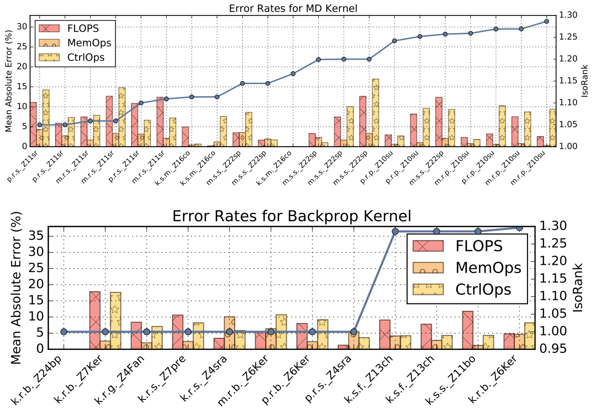

Fig. 3. Error rates when estimating instruction mixes statically from runtime observations for selected matched kernels (x-axis), with IsoRank scores near 1.30.

Figure 3 displays instruction mix estimation error rates, calculated using mean squared error, for MD and Backprop kernels as a function of matched kernels (x-axis) with IsoRank scores between 1.00 to 1.30. Naming convention for G_2 is as follows: $\langle$*architecture.suite.application.kernel*$\rangle$. In general, our approach is able to provide subgraph matching for arbitrary kernels through the IsoRank score in addition to instruction mixes within a 8% margin of error. Note that since relative dynamic performance is being estimated from static information, the error rates will always be high. By making use of subgraph matching strategy as well as instruction operations executed, our methodology is able to provide a mechanism to characterize unseen kernels.

V. Conclusion

We presented a control-flow-based methodology for analyzing the performance of CUDA applications. We combined static binary analysis with dynamic profiling to produce a set of metrics that not only characterizes the kernel by its computation requirements (memory or compute bound), but also provides detailed insights into application performance. Specifically, we provide an intuitive visualization and metrics display, and correlate performance hotspots with source line and file information, effectively guiding the end user to locations of interest and revealing potentially effective optimizations by identifying similarities of new implementations to known, autotuned computations through subgraph matching. We implemented this new methodology and demonstrated its capabilities on SHOC and Rodinia applications.

Future work includes incorporating memory reuse distance statistics of a kernel to characterize and help optimize the memory subsystem and compute/memory overlaps on the GPU. In addition, we want to generate robust models that will discover optimal block and thread sizes for CUDA kernels for specific input sizes without executing the application.