A Similarity Measure for GPU Kernel Subgraph Matching

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

Boyana Norris  
Computer and Information Science  
University of Oregon  
Eugene, OR, USA  
norris@cs.uoregon.edu

Allen D. Malony  
Computer and Information Science  
University of Oregon  
Eugene, OR, USA  
malony@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. CUDAflow is a tool that statically separates CUDA binaries into basic block regions and dynamically measures instruction and basic block frequencies. CUDAflow captures this information in a control flow graph (CFG) and performs subgraph matching across various kernel’s CFGs to gain insights to 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 CUDAflow is demonstrated with SHOC and Rodinia application case studies on a variety of GPU architectures, revealing novel thread divergence characteristics that facilitates end users, autotuners and compilers in generating high performing code.

Keywords

High performance computing, performance monitoring and evaluation, heterogeneous parallel programming.

1. INTRODUCTION

Thread divergence, which arises from the difficulty of mapping GPU programs onto SIMD (single instruction, multiple data) hardware units, presents a major challenge for applications that run on accelerated architectures (e.g. CUDA, OpenCL). When optimizing programs, compilers typically operate on the intermediate representation (IR) of a control flow graph, and not on the source code level, with the IR providing an abstraction from machine-specific intrinsics. A control flow graph (CFG) is a directed graph where nodes represent basic blocks of instructions and edges represent control flow paths. Identifying the base structured patterns of a CFG can shed light on the branch divergence problem [23], where a poorly written GPU kernel can severely impact execution performance. Structured programming consists of base constructs that represent how programs are written [8, 27]. Thus, compilers optimizing code will benefit from knowledge of the structuredness of a CFG, rather than the structuredness of a program.

Profiling-based approaches toward understanding thread divergence behavior [8, 13, 20, 21, 25] have been inefficient and lack scalability, since a flat profile not only consists of redundant information but also consumes unnecessary disk space per process $p$. Likewise, hardware performance counters have been prone to inaccuracies when monitoring divergence behavior [16, 17]. This research introduces how a control flow graph can capture sufficient information for a GPU kernel’s execution performance, which could enable code transformations by hooking onto a JIT compilation pass and executing the modified, high performing code.

To this end, we present CUDAflow, a scalable toolkit for heterogeneous computing applications. Specifically, CUDAflow provides a new methodology for characterizing CUDA kernels using control flow graphs and instruction operations executed and performs kernel subgraph matching to gain insights to 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 high performing code.

Contributions described in this paper include:

• Formalize control flow graphs for GPU kernels.
• Perform subgraph matching on various kernel CFG and GPUs.
• Reveal thread divergence behavior based on CFG properties.

The rest of the paper is organized as follows. Section 2 provides background information. Section 3 describes the methodology behind our CUDAflow tool and our implementation approach. Sections 4 and 5 summarizes the findings of our application characterization studies. Section 6 discusses prior work, and Section 7 outlines some directions for future work.

2. BACKGROUND

The next subsections cover preliminaries for structured programming and the thread divergence problem.

2.1 Structured Programming

Structured programming is a paradigm where programs are written using three base constructs: sequences of statements, if-then-else blocks, and loops. Figure 1 displays the base patterns in a graph representation. Compilers benefit from the structuredness of a CFG, including guaranteed reducibility for better compiler optimizations, ease of decompilation, and reduced performance degradation caused by thread divergence on SIMD units.

The following definitions refer to Figure 1 and will aid in further discussion.

• Path: A path between nodes $A$ and $B$ is an ordered list of adjacent edges and vertices, where the list begins with an out-edge $A$ and ends with an in-edge $B$.
• Condition node: Any node with two or more out-edges.

• Region: A region between two nodes (edges) $A$ and $B$ contains all nodes and edges present on any path from $A$ to $B$, where nodes are “internal” to a particular region.

• Dominator: A node (edge) $P$ is a dominator of node (edge) $Q$ if every path from the entry node of the CFG that reaches $Q$ has to pass through $P$.

• Post-dominator: A node (edge) $Q$ is a post-dominator of a node (edge) $P$ if every path from $P$ to the exit node of the CFG has to pass through $Q$.
  – Immediate post-dominator (IPDOM): Parent node of given node in post-dominator tree.
  – Dominator (post-dominator) relationships allow construction of a dominator (post-dominator) tree for the CFG.

• Single-entry-single-exit (SESE) region: A region between two nodes (edges) $A$ and $B$ is SESE if the following conditions hold: $A$ dominates $B$, $B$ post-dominates $A$, every cycle containing $A$ also contains $B$ and vice versa.

2.2 SIMD Thread Divergence

SIMD thread divergence originates from unstructured CFGs, further degrading thread execution performance on SIMD units. Unstructured CFGs are graphs that cannot be completely folded, and will be discussed in Section 2.2.4. For instance, consider the CFG in Figure 2 (left), where $P$ and $Q$ represent unstructured conditional nodes. For the two involved threads, $T_0$ and $T_1$, and for their execution pattern shown in Figure 2 (middle), condition $P$ behaves divergently, where the two threads execute different branch targets. Such divergent execution is handled using a mechanism called a reconvergence stack, where diverged threads resume joint execution only at the IPDOM of a divergent node $Q$. In addition, $R$ is computationally expensive and gets executed twice in this mechanism. The structured equivalent of CFG (Fig 2 right) however achieves combined execution of $R$ from both threads, lowering the overall execution time.

The above SASS assembly code illustrates how a control flow graph is constructed. Each basic block region is incremented with the number of times the node is visited. Upon sampling the program counter, the PC address is referenced internally to determine which basic block region the instruction corresponds to.

3. METHODOLOGY

In this section, our CUDAflow tool is presented whose overall organization is depicted in Figure 3. The dashed lines represent CUDAflow’s interaction with the current *nvcc* toolchain. Control flow graphs are constructed statically and the program counter is sampled dynamically, collecting counts of executed instructions and corresponding source code locations, among other information.

3.1 Parameter Space Characterization

This section describes the different components considered in the process of understanding control flow behavior in heterogeneous application codes.

3.1.1 Kernel Control Flow Graphs

One of the more complex parameters used to characterize SIMD thread divergence is with a control flow graph.

DEFINITION 1. 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.

Each basic block region is incremented with the number of times the node is visited. Upon sampling the program counter, the PC address is referenced internally to determine which basic block region the instruction corresponds to.

```
.L_41:
/*.04a0 */ DSETP, LE_AND P0, PT, 1 R61, + INF, PT;
/*.04c8 */ @P0 BRA *( L_43 ) ;
/*.04b0 */ LOP32I.OR R5, R7, 0x80000 ;
/*.04b8 */ MOV 4, R6 ;
/*.04c8 */ BRA *( L_42 ) ;
```

The above SASS assembly code illustrates how a control flow graph is constructed. Each basic block is labeled in the left margin (e.g. “L_41”), with branch instructions representing edges that lead to corresponding block regions (e.g. “L_43,” “L_42”). The PC offsets are listed in hexadecimal between the comments (/* */). In other words, “L_41” represents a node $n_i$ with “L_43” and “L_42” as its children.
Table 1: Control flow graphs for selected GPU kernels from SHOC (top) and Rodinia (bottom) application suite, comparing architecture variants.

| Kepler | Maxwell | Pascal | Kepler | Maxwell | Pascal | Kepler | Maxwell | Pascal |
|--------|---------|--------|--------|---------|--------|--------|---------|--------|
| ![Kepler Graph](image) | ![Maxwell Graph](image) | ![Pascal Graph](image) | ![Kepler Graph](image) | ![Maxwell Graph](image) | ![Pascal Graph](image) | ![Kepler Graph](image) | ![Maxwell Graph](image) | ![Pascal Graph](image) |
| BFS kernel_warp | Reduction reduce | SPMV csr_scalar | Hotspot calc_temp | Particlefilter sum_kernel | Pathfinder dynproc_kernel |

Example control flow graphs for selected SHOC (top) and Rodinia (bottom) GPU benchmarks are displayed in Table 1, comparing Kepler, Maxwell and Pascal architectures. Section 3.1 discusses the differences in GPU architectures. To simplify the discussion, we will refer to each GPU variant by its architecture family (Table 2, bottom). Notice how the CFG is slightly different for each architecture, due in part to the architecture layout of the GPU and its compute capability (NVIDIA virtual architecture). Although it may appear as if Maxwell uses fewer nodes for its CFGs, calc_temp actually uses one more node than its Kepler counterpart. Also, note that similarities in structure exist with several CFGs, including csr_scalar and sum_kernel. Part of the goal of this research is to predict the required resources for the application by inferring performance through CFG subgraph matching, with the subgraphs serving as building blocks for more nested and complex GPU kernels. Next, we introduce several metrics that build on this CFG representation.

3.1.2 Transition probability

Transition probabilities represent frequencies of an edge to a vertex, or of branches to code regions, which describes the application in a way that gets misconstrued in a flat profile. 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 assist compilers for code generation purposes.

**Definition 2.** A canonical adjacency matrix (CAM) is a square matrix $M$ that represents a graph $G$ such that every diagonal entry of $M$ is filled with the label of the corresponding node and every

![Figure 3: Overview of our proposed CUDAflow methodology.](image)
V, V′ is also referred to as a supergraph of G.

DEFINITION 4. A graph G = (V, E) is isomorphic to another graph G′ = (V′, E′) if and only if there exists a bijection f : V → V′ such that

- ∀u ∈ V, (f(u) = f′(f(u))).
- ∀u, v ∈ V, ((u, v) ∈ E ⇔ (f(u), f(v)) ∈ E′), and
- ∀(u, v) ∈ E, (f(u, v) = f′(f(u), f(v))).

DEFINITION 5. A graph G is subgraph isomorphic to a graph G′, denoted by G ⊑ G′, if and only if there exists a subgraph G″ of G′ such that G is isomorphic to G″.

DEFINITION 6. Given a set of graphs GD (referred as a graph database) and a threshold σ (0 < σ ≤ 1), the support of a graph G, denoted by supG is defined as the fraction of graphs in GD to which G is subgraph isomorphic.

$$\text{supG} = \frac{|\{G′ ∈ GD | G ⊑ G′\}|}{|GD|}$$

G is frequent if and only if supG ≥ σ.

The frequent subgraph mining problem is given a threshold σ and a graph database GD, finding all frequent subgraphs in GD.

In order to perform subgraph matching, we first scaled the matrices to the same size by taking for graphs G1 and G2 the maximal proper submatrix, constructed by B(Gi) = max(|V1|, |V2|) for a given Gi = min(|V1|, |V2|). The similarities in the shapes of the control flow graphs (Table 1) and the activity regions in the transition probability matrices (Fig. 4) motivated this approach. In our case, the dense hotspots in the transition matrix should align with its counterpart if the matrices are similar enough. Bilinear interpolation was used to scale the transition matrix before performing the pairwise comparison.

3.2 Bilinear Interpolation

Bilinear interpolation is an extension of linear interpolation for functions of two variables (e.g., x and y) on a rectilinear 2D grid. The idea is to perform linear interpolation first in one direction, and then again in the other direction. Interpolation works by using known data to estimate values at unknown points. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location.

DEFINITION 7. Suppose that we want to find the value of the unknown function f at the point (x, y). It is assumed that we know the value of f at the four points Q11 = (x1, y1), Q12 = (x1, y2), Q21 = (x2, y1), and Q22 = (x2, y2).

To find the value at P, for known values at Q, we first do linear interpolation in the x-direction. This yields

$$f(R_1) \approx \frac{x - x_1}{x_2 - x_1} f(Q_{11}) + \frac{x_2 - x}{x_2 - x_1} f(Q_{21})$$

$$f(R_2) \approx \frac{x - x_1}{x_2 - x_1} f(Q_{12}) + \frac{x_2 - x}{x_2 - x_1} f(Q_{22})$$

Then interpolate vertically

$$f(P) \approx \frac{y - y_1}{y_2 - y_1} f(R_1) + \frac{y_2 - y}{y_2 - y_1} f(R_2)$$

DEFINITION 8. Unit square: If we choose a coordinate system in which the four points where f is known are (0, 0), (0, 1), (1, 0), and (1, 1), then the interpolation formula simplifies to

$$f(x, y) \approx f(0, 0)(1-x)(1-y) + f(1, 0)x(1-y) + f(0, 1)(-1)x y + f(1, 1)xy$$
3.2.3 Pairwise Comparison

Once the matrix is interpolated, the affinity scores ($S_1$ and $S_2$ for graphs $G'_1$ and $G'_2$, respectively) are matched via a similarity measure, which includes the Kronecker product 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.

Kronecker product.

The Kronecker product, denoted by $\otimes$, is an operation on two matrices of arbitrary size resulting in a block matrix. It is a generalization of the outer product (which is denoted by the same symbol) from vectors to matrices, and gives the matrix of the tensor product with respect to a standard choice of basis.

If $A$ is a $m \times n$ matrix and $B$ is a $p \times q$ matrix, then the Kronecker product $A \otimes B$ is the $mp \times nq$ block matrix:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

Euclidean distance.

The Euclidean distance between points $p$ and $q$ is the length of the line segment connecting them ($\overline{pq}$). If $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$ are two points in Euclidean $n$-space, then the distance ($d$) from $p$ to $q$, or from $q$ to $p$ is given by the Pythagorean formula:

$$d(p, q) = d(q, p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

3.2.4 CFG Folding

This section discusses the steps involved in folding a control flow graph to determine whether the graph is structured or unstructured. The pseudocode is described in Algorithm 1 and detects for base structured patterns (sequence, selection, loop) defined in Section 2.1.

2.1 The first pass assigns to $N_i$ the root of the CFG. At each iteration, the base structured patterns are checked which entails conditions within itself. For instance, a child node’s count that is being evaluated would determine whether the subgraph is a sequence, a selection or a loop ...

Folding determines whether a control flow graph is structured and involves replacing the base structured pattern with a single node in the CFG. During folding, any edge not belonging to base pattern but has its source (or sink) node in base pattern is redirected so a newly created single node is its source (sink). Maximal folding repeatedly applies folding to a CFG until no more base structured patterns exist. Maximal folding operates in constant time $O(n)$, where $n$=#nodes in CFG. A CFG is structured if and only if the CFG is completely foldable; otherwise the CFG is unstructured.

3.3 Implementation

The methodology described in the previous section is implemented through a hybrid static and dynamic analysis approach. Refer to [18] for more information on the approach.

3.3.1 Hybrid static and dynamic analysis

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 instruction operations, program counter

---

**Table 2: Graphical processors used in this experiment.**

|          | K80 | M40 | P100 |
|----------|-----|-----|------|
| CUDA capability | 3.5 | 5.2 | 6.0  |
| Global memory (MB) | 11520 | 12288 | 16276 |
| Multiprocessors (MP) | 13 | 24 | 56  |
| CUDA cores per MP | 192 | 128 | 64  |
| CUDA cores | 2496 | 3072 | 3584 |
| GPU clock rate (MHz) | 824 | 1140 | 405  |
| Memory clock rate (MHz) | 2505 | 5000 | 715  |
| L2 cache size (MB) | 1.572 | 3.146 | 4.194 |
| Constant memory (bytes) | 65536 | 65536 | 65536 |
| Shared mem blk (bytes) | 49152 | 49152 | 49152 |
| Registers per block | 65536 | 65536 | 65536 |
| Warp size | 32 | 32 | 32  |
| Max threads per MP | 2048 | 2048 | 2048 |
| Max threads per block | 1024 | 1024 | 1024 |
| CPU (Intel) | Haswell | Ivy Bridge | Haswell |

**Table 3: Hardware and execution environment information.**

| Architecture | Haswell | Ivy Bridge |
|--------------|---------|------------|
| Model        | E5-2698 v3 | Xeon X5650 |
| Clock speed  | 2.30 GHz | 2.67 GHz  |
| Node count   | 4, 14   | 6         |
| GPUs         | 4xK80, 4xP100 | 3xM40 |
| Memory       | 256 GB  | 50 GB     |
| Linux kernel | 3.10.0-229.14.1 | 2.6.32-642.4.2 |
| Compiler     | CUDA v8.0.27 |
| Flags        | \{'g', 'lineinfo', 'arch=sm_cc'\} |
offsets, and line information. The CFG structure was stored in iGraph format [6]. We attribute the static analysis from the objdump file to the profiles collected from the source code activity to provide runtime characterization 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.

4. EXPERIMENTAL SETUP

This section describes the execution environment and the applications as part of this work.

4.1 Execution environment

To demonstrate our proposed CUDAflow methodology, we measured the performance of applications on a variety of GPU architectures. The hardware architecture and software environment configuration are displayed in Table 3, whereas the graphic processor units are listed in Table 2. Table 3 corresponds to the CPU entry in Table 2. The selected GPUs reflect the various architecture family generations, and performance results presented in this paper represent GPUs belonging to the same family. For instance, we observed that the performance results from a K80 architecture and a K40 (both Kepler) were similar, and, as a result, did not include comparisons of GPU architectures within families. Also, note the changes in architectural features across generations (global memory, MP, CUDA cores per MP), as well as ones that remain fixed (constant memory, warp size, registers per block). For instance, while the number of multiprocessors increased in successive generations, the number of CUDA cores per MP (or streaming multiprocessors, SM) actually decreased. Consequently, the number of CUDA cores (MP $\times$ CUDA cores_per_mp) increased in successive GPU generations.

4.2 Applications

Rodinia and SHOC application suite are a class of GPU applications that cover a wide range of computational patterns typically seen in parallel computing. Table 4 provides a description of the applications used in this experiment along with source code statistics, including the number of kernel functions, the number of associated files and the total lines of code.

4.2.1 Rodinia

Rodinia is a benchmark suite for heterogeneous computing to help architects study emerging platforms such as GPUs (Graphics Processing Units), which includes applications and kernels that target multi-core CPU and GPU platforms [5]. Rodinia covers a wide range of parallel communication patterns, synchronization techniques and power consumption, and has led to architectural insights such as memory-bandwidth limitations and the consequent importance of data layout.

4.2.2 SHOC Benchmark Suite

The Scalable Heterogeneous Computing (SHOC) application suite is a collection of benchmark programs testing the performance and stability of systems using computing devices with non-traditional architectures for general purpose computing [7]. SHOC provides implementation versions for CUDA, OpenCL, and Intel MIC, and supports execution on clusters with MPI as well as single-node hosts.

5. ANALYSIS

We analyzed the SHOC and Rodinia applications using our new methodology and discuss the results at different granularities.
Table 4: Description of SHOC and Rodinia benchmarks used in this work.

| Name           | Kernels | Files | Lines | Description                                                                 |
|----------------|---------|-------|-------|------------------------------------------------------------------------------|
| FFT            | 9       | 4     | 970   | Forward and reverse 1D fast Fourier transform.                               |
| MD             | 2       | 2     | 717   | Computation of the Lennard-Jones potential from molecular dynamics.          |
| MD5Hash        | 1       | 1     | 720   | Computation of many small MD5 digests, heavily dependent on bitwise operations.|
| Reduction      | 2       | 5     | 785   | Reduction operation on an array of single or double precision floating point values.|
| Scan           | 6       | 6     | 1035  | Scan (parallel prefix sum) on an array of single or double precision floating point values.|
| SPMV           | 8       | 2     | 830   | Sparse matrix-vector multiplication.                                        |
| Stencil2D      | 2       | 12    | 1487  | A 9-point stencil operation applied to a 2D dataset.                        |
| Backprop       | 2       | 7     | 945   | Trains weights of connecting nodes on a layered neural network.             |
| BFS            | 2       | 3     | 971   | Breadth-first search, a common graph traversal.                             |
| Gaussian       | 2       | 1     | 1564  | Gaussian elimination for a system of linear equations.                      |
| Heartwall      | 1       | 4     | 6017  | Tracks changing shape of walls of a mouse heart over a sequence of ultrasound images.|
| Hotspot        | 1       | 1     | 1199  | Estimate processor temperature based on floor plan and simulated power measurements.|
| Nearest Neighbor| 1      | 2     | 385   | Finds k-nearest neighbors from unstructured data set using Euclidean distance.|
| Needleman-Wunsch| 2     | 3     | 1878  | Global optimization method for DNA sequence alignment.                      |
| Particle Filter| 4       | 2     | 7211  | Estimate location of target object given noisy measurements in a Bayesian framework.|
| Pathfinder      | 1       | 1     | 707   | Scan (parallel prefix sum) on an array of single or double precision floating point values.|
| SRAD v1        | 6       | 12    | 3691  | Diffusion method for ultrasonic and radar imaging applications based on PDEs.|
| SRAD v2        | 2       | 3     | 2021  | ...                                                                          |

Table 5: Benchmark runs (ms), comparing architectures.

| Name           | K80     | M40    | P100    |
|----------------|---------|--------|---------|
| FFT            | 774808.25 | 1032196.25 | 932807 |
| MD             | 4941300.25 | 550667.25  | 5448244.75 |
| MD5Hash        | 7258343.75 | 4422818.5  | 2281888.5 |
| Reduction      | 42448007  | 4805355.3  | 41773217.25 |
| Scan           | 13643441.5 | 14987326.5  | 139658287.75 |
| SPMV           | 93229082  | 108585496.25 | 99209322.25 |
| Stencil2D      | 331556520.5 | 741662177.75 | 1135803566.5 |
| Backprop       | 41900.25  | 58432.5  | 37779.25 |
| BFS            | 452606.75 | 364812.25 | 390988.25 |
| Gaussian       | 246234.75 | 282416  | 258588 |
| Heartwall      | 35475401.5 | 51792633  | 52117034.25 |
| Hotspot        | 15062.75  | 12649   | 8045.25 |
| NN             | 253.25    | 284.25  | 174   |
| NW             | 3682012.25 | 6482659.75 | 6217623.25 |
| Particlefilter  | 854584.75 | 1849342.75 | 2125817.5 |
| Pathfinder      | 979597.75 | 938474.25 | 766344.25 |
| srad_v1        | 6926474.75 | 8959897.5  | 8161951.25 |
| srad_v2        | 451102.5  | 409342.75 | 267043.5 |

5.1 Application level

Table 5 displays total execution time for all applications, comparing different architectures. The execution readings were made with the TAU Performance System. Note that run times for each application was somewhat similar across architectures, although the Maxwell GPU performed poorly for all applications except for BFS. Figure 5 projects goodness as a function of efficiency (bottom figure zooms in on dense region of top figure), which displays the similarities and differences of the benchmark applications. The size of bubble represents the number of operations executed, whereas the shade represents the GPU type. Efficiency describes how gainfully employed the GPU floating-point units remained, or flops per second:

\[
\text{efficiency} = \frac{\text{op}_{\text{flop}} + \text{op}_{\text{fost}} + \text{op}_{\text{send}} + \text{op}_{\text{recv}}}{\text{time}_{\text{exec}}} \cdot \text{calls}_n \quad (1)
\]

Goodness metric describes the intensity of the floating-point and memory operation arithmetic intensity:

\[
\text{goodness} = \sum_{j=1}^{n} \text{op}_j \cdot \text{calls}_n \quad (2)
\]

Figure 5 shows a positive correlation between the two measures, where the efficiency of an application increases along with its goodness.

5.2 CFG subgraph matching

Table 6 provides statistics for various metrics from CUDAflow for randomly selected kernels, comparing Kronecker and Euclidean CFG subgraph matching strategies, with score representing the similarity of the CFGs. The naming convention adopted for each kernel is as follows: `<architecture.suite.application.kernel>`. CFG properties are also displayed with the number of vertices and edges representing V and E, respectively. The instruction mix percentages are kernel-specific, with the instruction mix values normalized according to its total instruction count. The selected kernels not only display the diverseness of the kernels, with V and E varying between the two as well as it operations intensity, but also the performance of each measure regardless of application or architecture. For instance, SPMV and ParticleFilter kernels (Table 6, (1) and (2, 2)) scored 1.11 (Table 4, 2nd row), which demonstrates that...
Table 6: Breakdown statistics of our CFG subgraph matching algorithm for selected GPU benchmark kernel combinations.

| GPU.Suite.Kernel | $G_1$ | $V_1$ | $E_1$ | $FL_1$ | $MEM_1$ | $CTL_1$ | GPU.Suite.Kernel | $G_2$ | $V_2$ | $E_2$ | $FL_2$ | $MEM_2$ | $CTL_2$ | Similarity | Kron | Eucl |
|------------------|-------|-------|-------|--------|---------|---------|------------------|-------|-------|-------|--------|---------|---------|-------------|------|------|
| m.s.md.Z16com    | 3     | 4     | 51.1  | 10.6   | 34.1    |         | m.s.md.Z16com    | 3     | 4     | 51.1  | 10.6   | 34.1    |         | 1.00        | 1.00 | 1.00 |
| m.s.spmv.Z22spmv | 11    | 6     | 51.9  | 16.9   | 25.9    |         | m.r.particlef.Z10sum | 14   | 6     | 36.2  | 12.1   | 50.0    |         | 1.11        | 1.74 | 1.11 |
| k.s.md.Z16com    | 8     | 5     | 42.7  | 13.2   | 29.4    |         | m.r.particlef.Z10sum | 14   | 6     | 37.5  | 12.5   | 48.2    |         | 1.28        | 2.01 | 1.28 |
| p.pathfinder.Z14dy | 4    | 4     | 60.0  | 8.9    | 31.1    |         | m.s.md.Z16com    | 3     | 4     | 51.1  | 10.6   | 34.1    |         | 1.35        | 1.35 | 1.35 |
| k.r.bfs.Z6Kernel | 5     | 3     | 65.0  | 12.5   | 20.0    |         | p.r.srad_v2.Z11rad | 4     | 5     | 64.4  | 19.5   | 16.1    |         | 1.43        | 1.43 | 1.43 |
| p.s.md.Z16com    | 7     | 6     | 56.1  | 7.1    | 33.3    |         | m.s.spmv.Z22spmv | 2     | 5     | 67.5  | 17.5   | 10.7    |         | 1.53        | 1.80 | 1.53 |
| k.s.scan.Z6reduc | 13    | 10    | 54.1  | 25.6   | 14.3    |         | m.r.particlef.Z14calc | 16   | 10    | 63.9  | 9.1    | 27.1    |         | 1.65        | 1.87 | 1.65 |
| k.s.fft.Z13chk512 | 2    | 3     | 56.4  | 30.1   | 12.1    |         | p.r.backprop.Z22bpnn | 3     | 3     | 78.1  | 8.5    | 13.4    |         | 1.75        | 1.79 | 1.75 |
| p.s.md.Z16com    | 7     | 6     | 56.1  | 7.0    | 33.3    |         | k.r.s.md.Z16com   | 3     | 4     | 51.1  | 10.6   | 34.1    |         | 1.82        | 2.13 | 1.82 |
| p.nw.Z20need     | 21    | 21    | 44.6  | 32.3   | 20.6    |         | m.s.stencil2d.Z13St | 6     | 11    | 76.7  | 14.4   | 8.6     |         | 1.90        | 1.90 | 1.90 |

Table 7: Similarity score of kernels when measured against itself, comparing GPU architectures.

|                   | Similarity | Kepler | Maxwell | Pascal |
|-------------------|------------|--------|---------|--------|
| BFS               |            | 1.04   | 1.29    | 1.29   |
|                   |            | 1.29   | 1.00    | 1.00   |
|                   |            | 1.58   | 1.94    | 1.94   |
|                   |            | 1.94   | 1.59    | 1.45   |
|                   |            | 1.95   | 1.60    | 1.47   |
| Hotspot           |            | 1.33   | 2.64    | 2.64   |
|                   |            | 2.64   | 1.51    | 1.51   |
|                   |            | 1.31   | 1.12    | 1.11   |
|                   |            | 1.31   | 1.12    | 1.10   |
|                   |            | 1.53   | 1.31    | 1.29   |
|                   |            | 1.53   | 1.31    | 1.29   |
|                   |            | 1.56   | 1.93    | 1.92   |
|                   |            | 1.93   | 1.50    | 1.29   |
|                   |            | 1.92   | 1.50    | 1.54   |
| MD                |            | 1.18   | 1.43    | 1.41   |
|                   |            | 1.43   | 1.35    | 1.71   |
|                   |            | 1.12   | 1.71    | 1.34   |
| Scan              |            | 1.00   | 1.99    | 2.00   |
|                   |            | 1.00   | 1.99    | 2.00   |
|                   |            | 1.00   | 1.99    | 2.00   |
| SPMV              |            | 2.25   | 2.79    | 2.39   |
|                   |            | 2.79   | 1.66    | 2.26   |
|                   |            | 2.39   | 2.26    | 1.67   |

Figure 6: Differences in vertices between two graphs, as a function of Kronecker metric for all GPU kernel combinations. Color represents intensity.
Control flow divergence in heterogeneous computing applications is a well known and difficult problem, due to the lockstep nature of the GPU execution paradigm. Current efforts to address branch divergence in GPUs draw from several fields, including profiling techniques in CPUs, and software and hardware architectural support in GPUs. For instance, Sarkar demonstrated that the overall execution time of a program can be estimated by deriving the variances of basic block regions [24]. Control flow graphs for flow and context sensitive profiling were discussed in [1, 2], where instrumentation probes were inserted at selected edges in the CFG, which reduced the overall profiling overhead with minimal loss of information. Hammock graphs were constructed [30] that mapped unstructured control flow on a GPU [9, 28]. By creating thread frontiers to identify early thread reconvergence opportunities, dynamic instruction counts were reduced by as much as 633.2%.

Both hardware and software approaches exist to address the control flow divergence problem. For hardware, thread-aware predication code aided the CUDA compiler in systematically mapping nested control flow structures to data-parallel architectures [15]. Similarly, a dual-path stack architecture approach increased path parallelism using GPGPU-sim, which traversed depth-first on the control flow tree followed by a single path traversal [22].

The authors [10] have characterized PTX kernels by creating an internal representation of a program and running it on an emulator, which determines the memory, control flow and parallelism of the application. This work closely resembles ours, but differs in that we perform workload characterization on actual hardware during execution. The MIAMI toolkit [19] is an instrumentation framework for studying an application’s dynamic instruction mix and control flow, but does not include support for GPUs.

Subgraph matching has been explored in a variety of contexts. For instance, the DeltaCon framework matched arbitrary subgraphs based on similarity scores [14], which exploited the properties of the graph (e.g. clique, cycle, star, barbell) to support the graph matching. Similarly, frequent subgraph mining was performed on molecular fragments for drug discovery [4], whereas document clustering was formalized in a graph database context [12]. However, none of these approaches apply frequent subgraph matching for code generation purposes.

7. CONCLUSION

We have presented CUDAflow, 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, whether memory or compute bound, but also provides pinpointed and detailed insights in 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. We have 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. Last, we are in the process of developing an online web portal [26] that will archive a collection of control flow graphs for all known GPU applications. For instance, the web portal would be able to make on-the-fly comparisons across various hardware resources, as well as other GPU kernels, without burdening the end user with hardware requirements or software package installations, and will enable more feature rich capabilities when reporting performance metrics.

8. REFERENCES
Figure 8: Kronecker similarity measure for selected kernels with respect to all other GPU kernels from SHOC and Rodinia suite, comparing architecture types.

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[3] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[4] C. Borgelt and M. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Data Mining*, 2002. *ICDM 2003. Proceedings*. 2002 IEEE International Conference on, pages 51–58. IEEE, 2002.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[6] G. Csardi and T. Nepusz. The iGraph software package for complex network research.

[7] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, B. Norris, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[8] Allinea DDT, [http://www.allinea.com/products/ddt](http://www.allinea.com/products/ddt).

[9] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 477–488. ACM, 2011.

[10] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. In *Performance Analysis of Systems and Software (ISPASS)*, 2012 IEEE International Symposium on, pages 58–67. IEEE, 2012.

[11] R. C. Gonzales and R. E. Woods. Digital Image Processing. Addison-Wesley, 1993.

[12] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining*, 2003. *ICDM 2003. Third IEEE International Conference on*, pages 549–552. IEEE, 2003.

[13] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[14] D. Koutra, J. T. Vogelstein, and C. Faloutsos. DeltaCon: A principled massive-graph similarity function. SIAM.

[15] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanovic. Exploring the design space of SPMD divergence management on data-parallel architectures. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 101–113. IEEE, 2014.

[16] R. Lim, D. Carrillo-Cisneros, W. Allkowialet, and I. Scherson. Computationally efficient multiplexing of events on hardware counters. In *Linux Symposium*, 2014.

[17] R. Lim, A. Malony, B. Norris, and N. Chaimov. Identifying optimization opportunities within kernel execution in CPU codes. In *Euro-Par 2015: Parallel Processing Workshops*. Springer, 2015.

[18] R. Lim, B. Norris, and A. Malony. Tuning heterogeneous computing architectures through integrated performance tools. In *GPU Technology Conference*, page poster, 2016.

[19] G. Marin, J. Dongarra, and D. Terpstra. Miami: A framework for application performance diagnosis. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 158–168. IEEE, 2014.

[20] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 2–13. ACM, 2004.

[21] Nvidia Visual Profiler, [https://developer.nvidia.com/nvidia-visual-profiler](https://developer.nvidia.com/nvidia-visual-profiler).

[22] M. Rhu and M. Erez. The dual-path execution model for efficient GPU control flow. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 591–602. IEEE, 2013.

[23] A. Sabne, P. Sakdhnagool, and R. Eigenmann. Formalizing structured control flow graphs. In *Languages and Compilers for Parallel Computing (LCPC)*, volume 10136. Lecture Notes in Computer Science, 2016.

[24] V. Sarkar. Determining average program execution times and their variance. In *ACM SIGPLAN Notices*, volume 24, pages 298–312. ACM, 1989.
[25] S. S. Shende and A. D. Malony. The TAU parallel performance system. \textit{International Journal of High Performance Computing Applications}, 20(2):287–311, 2006.

[26] S. Sreepathi, M. Grodowitz, R. Lim, P. Taffet, P. C. Roth, J. Meredith, S. Lee, D. Li, and J. Vetter. Application characterization using Oxbow toolkit and PADS infrastructure. In \textit{Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing}, pages 55–63. IEEE Press, 2014.

[27] M. H. Williams and H. Ossher. Conversion of unstructured flow diagrams to structured form. \textit{The Computer Journal}, 21(2):161–167, 1978.

[28] H. Wu, G. Diamos, S. Li, and S. Yalamanchili. Characterization and transformation of unstructured control flow in GPU applications. In \textit{1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems}, 2011.

[29] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In \textit{Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on}, pages 721–724. IEEE, 2002.

[30] F. Zhang and E. H. D’Hollander. Using hammock graphs to structure programs. \textit{IEEE Transactions on Software Engineering}, 30(4):231–245, 2004.