A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation

Xiebing Wang*, Kai Huang†, Alois Knoll* and Xuehai Qian‡
*Department of Informatics, Technical University of Munich, Munich, Germany
†School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China
‡Department of Computer Science, University of Southern California, Los Angles, USA
*{wangxie, knoll}@in.tum.de, †huangk36@mail.sysu.edu.cn, ‡xuehai.qian@usc.edu

***Abstract*—This paper proposes a hybrid framework for fast and accurate performance estimation of OpenCL kernels running on GPUs. The kernel execution flow is statically analyzed and thereupon the execution trace is generated via a loop-based bidirectional branch search. Then the trace is dynamically simulated to perform a dummy execution of the kernel to obtain the estimated time. The framework does not rely on profiling or measurement results which are used in conventional performance estimation techniques. Moreover, the lightweight trace-based simulation consumes much less time than a fine-grained GPU simulator. Our framework can accurately grasp the variation trend of the execution time in the design space and robustly predict the performance of the kernels across two generations of recent Nvidia GPU architectures. Experiments on four Commercial Off-The-Shelf (COTS) GPUs show that our framework can predict the runtime performance with average Mean Absolute Percentage Error (MAPE) of 17.04% and time consumption of a few seconds. We also demonstrate the practicability of our framework with a real-world application.**

I. Introduction

To fully exploit the computing power of GPU, program developers need a deep understanding of its parallel working mechanism, in order to efficiently process the workload at runtime. This poses a challenge for non-expert users because they have no prior knowledge about elaborate parallel programming. To solve this, two approaches, namely auto-tuning and performance estimation, are used to help seek the optimal execution from the vast program design space. Traditional auto-tuning searches through either the whole [1] [2] or a sliced [3] [4] design space, which causes a considerable amount of time. Although this time cost can be reduced by optimization [5] or machine learning based algorithms [6], the relevance between the program input configuration and the resulted performance gain still remains obscure. Therefore, performance estimation is essential to crack the internal program runtime behavior so as to improve the external program execution efficiency.

State-of-the-art GPU performance estimation still suffers from several constraints. First, performance model always needs to be subtly tuned for the appropriate configurations of the target program to obtain convincing estimations. This makes it rather difficult to derive a general-purpose instead of application-oriented method. Secondly, performance estimation approaches can hardly keep up with the rapid architectural change of contemporary GPUs, due to the continuously promotion and upgrade of Commercial Off-The-Shelf (COTS) products. Although machine learning based methods [7] [8] [9] are applicable to general platforms, the off-line feature sampling of the hardware counter metrics over the huge design space incurs a significant amount of time and the trained model is sensitive to unknown applications. Last but not least, there still exists possibility to improve the accuracy and usability of state-of-the-art GPU performance models [10]. Although fine-grained GPU simulators could give rather accurate estimations, the extremely large time consumption makes it unsuitable for practical use [11] [12].

To address the aforementioned issues, this paper proposes a hybrid framework to estimate the performance of parallel applications on the GPU. We target OpenCL [13] workload because OpenCL is a cross-platform standard and therefore the proposed framework can still be applied to other accelerators. The high-level kernel source code is first transformed into LLVM [14] Intermediate Representation (IR) instructions, from which the program execution trace is generated based on GPU's philosophy of parallelism. We developed a lightweight simulator to dynamically consume the arithmetic and memory access operations in the execution trace in granularity of 32 work items or so-called warps. The hardware specification and micro-benchmarking metrics are also fed to this simulator to obtain the estimated execution time.

In contrast to conventional analytical or machine learning based methods, our framework does not require extra hardware performance counter metrics captured by a third-party profiler, or measurement results which are obtained after executing the whole or a portion of the target kernel before the estimation. Meanwhile, unlike fine-grained GPU simulators that spend simulation time in the scale of hours [15] [16], our framework can give estimation results in a few seconds. For the evaluation, we validate our framework with 20 different kernels from the Rodinia [17] benchmark. We conduct a design space exploration of all possible input parameter combinations which counts to in total 306,558 cases. Experiments on four COTS Nvidia GPUs across two architectures (Kepler and Maxwell) show that our framework can accurately grasp the variation trend of the kernel execution time in the design space, which indicates that our framework can also be utilized to find the optimal execution in the vast design space. On average, the proposed framework can give performance estimation with Mean Absolute Percentage Error (MAPE) of 17.04%. Moreover,

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00062

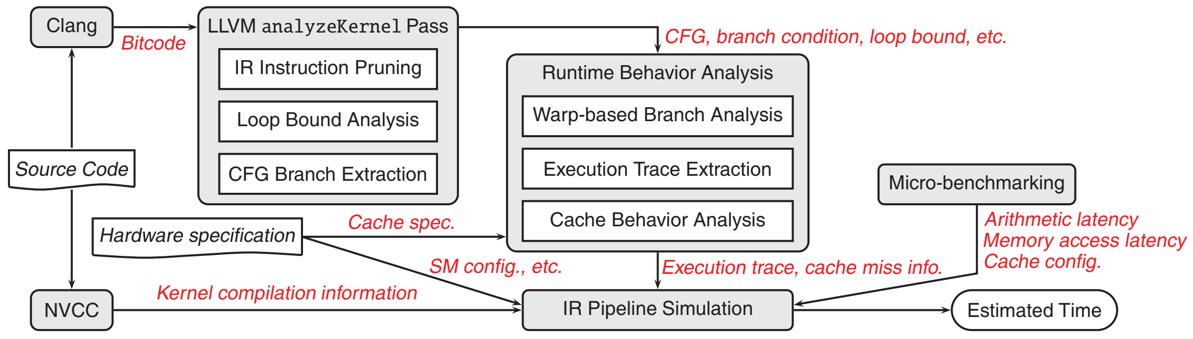


Figure 1: Overview of the performance estimation framework.

the case study on a real-world lane detection application achieves prediction accuracy with MAPE of 17.38%. The contributions of this paper are as follows:

- We propose a hybrid framework that combines source-level analysis and trace-based simulation to predict the performance of GPU kernels. The execution trace of the target kernel is statically generated and then simulated to estimate the runtime performance.
- We propose a loop-based bidirectional branch search algorithm to extract the kernel execution trace that models the warp execution flow of the GPU kernel.
- We develop a lightweight simulator to mimic the kernel execution and then predict the runtime performance results, taking into consideration both the IR instruction pipeline and cache modeling. The simulator can accurately predict the performance of kernels running across different GPU platforms in a few seconds.
- We demonstrate the accuracy and practicability of our framework with the Rodinia [17] benchmark and a real-world application, on four Nvidia GPUs across two generations of recent architectures.

The remainder of this paper is organized as follows: Section II gives overview of the proposed framework. Section III and Section IV presents the source-level analysis and trace-based simulation, respectively. Section V gives experimental analysis and Section VI presents a lane detection case study. Section VII is related work and Section VIII concludes.

II. Framework Overview

Figure 1 gives the overview of the proposed performance estimation framework. The kernel source code is first processed by Clang compiler to generate the LLVM bitcode file that contains IR instructions of the target kernel. Meanwhile, the source file is passed to NVCC compiler to obtain kernel compilation information that includes the detailed runtime resource usage of the kernel, such as the number of used on-chip registers and used shared memory size. The framework mainly contains two modules, i.e. the source-level analysis and the subsequent trace-based simulation. In the source-level analysis module, the kernel bitcode file is processed by an LLVM `analyzeKernel` pass and the execution trace is subsequently extracted from the kernel runtime behavior analysis. The `analyzeKernel` pass prunes IR instructions in the basic blocks so that only the arithmetic and memory access operations, which contribute to the kernel execution time, are retained. The execution flow information, such as the loop statements and the branches, is extracted and analyzed for the following execution trace generation.

Given the Control Flow Graph (CFG) and the execution flow information, the kernel runtime behavior is then analyzed and the execution trace is generated in granularity of warps. The cache miss/hit information is subsequently obtained according to the cache specification and the execution trace. The simulation module mimics the kernel runtime behavior by virtue of constructing an IR instruction pipeline and consuming the execution trace iteratively. A set of micro-benchmarks are used to calibrate the target GPU to obtain the hardware metrics such as latencies of the arithmetic operations, latencies of the memory access operations, and the cache configurations. These hardware metrics, together with the hardware specification, the kernel compilation information, the kernel execution trace, and the cache miss information, are fed to the simulator to estimate the final execution time.

III. Source-level Analysis

A. LLVM `analyzeKernel` pass

The `analyzeKernel` pass collects the basic blocks and builds the CFG of the target kernel. For each basic block, we document the IR instructions to construct the intra-block execution trace. The execution flow information used to generate the execution trace is obtained via the three steps illustrated as follows.

1) IR instruction pruning: We assume that the execution time is mainly consumed by the arithmetic and memory access operations. Therefore for each basic block, we filter out the time-cost-irrelevant instructions such as the LLVM-specific intrinsic annotations `llvm.lifetime.start`, `llvm.lifetime.end`, the memory address calculation instruction `getelementptr`, the data type conversion instructions `trunc`, `ext`, and so on. Note that here these instructions are only removed from the execution trace, but are still used for the later control flow analysis.

As for function calls, we observe that the `call` instruction appears only when invoking ① the OpenCL work-item built-in functions, such as `get_global_id`, `get_local_id`, etc., ② the synchronization function `barrier`, or ③ the LLVM intrinsic functions such as `llvm.fmuladd.f32`, etc. The subfunctions in the source code are replaced by detailed instructions and therefore non-existent in the bitcode file. Consequently, we record all the related information about these function calls and this information is used to assist the execution trace generation whenever necessary. The OpenCL work-item built-in functions are highlighted because their return values typically serve as memory address indices that directly determine the memory access pattern. The synchronization function is labelled as a flag that notifies the wait signal of the warp execution in the pipeline. The LLVM intrinsic functions are also converted to the corresponding arithmetic operations in the kernel execution trace.

2) Loop bound analysis: Instead of deducing a precise value of the loop trip count, we attempt to estimate the loop bound of each basic block in the loop. The reasons are multifold. First, state-of-the-art static loop analysis is still an open problem [18] and therefore it is impossible to adopt a generic method to obtain the loop trip count of arbitrary code blocks. Secondly, in general, the input of parallel applications is a regular rectangle- or cuboid-like grid that can be ideally decomposed and mapped to the threads on the GPU. The formation of the high-level loop code is regular in the majority of the cases. Lastly, the loop bound manifests an extreme case of the execution of the loop and this scenario should also be considered when analyzing the performance of the kernel executions.

We first use Loopus [19] to analyze the loop bound. We observe that Loopus can handle simple loops, i.e., when the loop induction variable is a fixed constant. For more complicated loops, the loop bound is first determined by performing an LLVM Scalar Evolution (SE) analysis [20] of the basic blocks in the loop. The SE analysis gives an explicit bound if the target basic block either is within a single-exit loop or has a predictable backedge taken count.

When both Loopus and LLVM SE analysis fail to give outputs, an extra static analysis of the loops is performed to further extract the loop bound. The main idea of this static analysis is to identify the loop induction variable and track its value at the scope of the entire kernel function. First, the exit basic blocks of the loop are collected, from which the true exit basic block is set as the loop latch block. The terminator of the true exit basic block is the loop induction instruction and we observe that for all the test kernels this instruction is a conditional branch form of a `br` instruction. The conditional branch has two arguments, of which the first is either the loop induction variable or the loop induction variable updated with a increment of the loop step size, and the second is the end value, which is loop invariant, of the loop induction variable. In LLVM, the loop induction variable is represented as a Static Single Assignment (SSA) and this SSA could be: ① binary operation such as `add`, `mul`, etc. ② `load` instruction. ③ `phi` instruction. In case ①, we traverse all the `phi` nodes in the loop header block, from which the loop induction variable is set as the `phi` node of which the return value equals the updated loop induction variable, when taking the loop latch block as the incoming value. With regards to case ②, we track all the `store` instructions that write data to the pointer argument of this `load` instruction, by virtue of the memory dependency analysis. The memory write value of the `store` instruction that lies outside of and closest to the loop is deemed the start value, which is also loop invariant, of the loop induction variable. For case ③, we also traverse all the `phi` nodes in the loop header block and extract the `phi` node which equals the loop induction instruction. Then the updated value of the loop induction variable equals the return value of this `phi` node when taking the loop latch block as the incoming value. With the start value, the end value, and the step size of the loop induction variable obtained, the loop bound is calculated as the induction time of the loop induction variable within the loop: $loopBound = \frac{endValue - startValue}{stepSize}$.

With regards to the nest loops, the analyzed result only indicates the loop bound of the basic block at its current loop level and each of the outer loop bound values equals the loop bound value of one of the preceding basic blocks, which lies exactly at its corresponding loop level. For each basic block in the nest loop, at each upper loop level, we record its closest preceding basic block so that the nest loop chain is maintained, for ease of the later execution trace generation. If the deduced loop bound relies on the induction variable of the outer loop, then we record the different loop bound values when the outer loop iterates. During our experiments, the aforementioned static analysis manages to give the loop bound of all the loop basic blocks in the test kernels.

3) CFG branch extraction: We extract the triggering condition of each branch by analyzing the `phi` and `br` instructions within the head and tail basic blocks of that branch path. The `br` instruction is associated with a `cmp` instruction from which we can deduce the branch condition. The branch condition is an expression that contains the logical operation combination of several variables of which some are conditional variables and the other are constants. The conditional variable is represented as an SSA and it can be further refined with one or more SSAs associated with it. This is done by an iterative search, which terminates when the termination SSA is: ① a kernel argument. ② a temporary variable. ③ a memory load of the data pointed by a kernel argument, which is a pointer parameter.

B. Runtime Behavior Analysis

1) Warp-based branch analysis: To determine whether a branch condition is hit or miss, we evaluate the execution of the branch paths in granularity of warps. As shown in Section III-A3, the values of the branch conditional variables can be classified into three cases. For case ①, this branch path is easily determined to be hit or miss since the input kernel arguments are known. In case ②, if the temporary variable is thread-ID-dependent, i.e., the variable is the return value of the aforementioned OpenCL work-item built-in functions, then this branch path can also be determined to be hit or miss, given the warp ID and the global and local work size configuration of the target kernel. If the temporary

Algorithm 1: Execution Trace Generation

```
Input: CFG Entry Node B, CFG Exit Node E, Backedge Set $\mathbb{BE}$,
       Non-backedge Set $\mathbb{NE}$, Basic Block Data Description List
       BBInfoList, Warp ID wid, Mask Array $M$
Output: Kernel Execution Trace $\mathbf{T}$
1  $\mathbf{T} \leftarrow \varnothing$, $\mathbb{T} \leftarrow \varnothing$, $\tau \leftarrow$ B   ▷ Initialize the execution trace with entry node B
2  updateExecTrace($\tau$, $\mathbb{T}$, BBInfoList, wid, $M$)
3  ST $\leftarrow \varnothing$ ▷ Initialize a stack to store the header nodes of multiple branch paths
4  while $\tau \neq$ E $\wedge \neg$ E.isVisited do ▷ Terminate when exit node is visited
5      $\tau \leftarrow$ getTraceSuccNode($\tau$, B, BBInfoList, ST, $\mathbb{T}$, $\mathbb{BE}$, $\mathbb{NE}$)
6      updateExecTrace($\tau$, $\mathbb{T}$, BBInfoList, wid, $M$)
7  for $i \leftarrow 0$ to $\mathbb{T}$.size() do
8      if $M[i]! = 0$ then ▷ Remove branch miss nodes from the generated trace
9          $\mathbf{T} \leftarrow \mathbf{T} + \{\mathbb{T}.at(i)\}$
10 return $\mathbf{T}$
```

variable is the loop induction variable, we can also mask or unmask this branch path, depending on the logical result of the branch condition at different loop iterations. For the remaining cases we assume this branch path is always hit. For case ③, because the value of this memory load can only be determined at runtime, for the sake of conservation we also assume this branch path is always hit.

2) Execution trace generation: Let's first consider how GPU walks along the CFG to execute the kernel. For Nvidia GPUs, each OpenCL work item instance is mapped to a thread and a group of 32 threads are bound together to execute the instance in lock-step manner. This group of threads is called a warp for Nvidia GPUs and the counterpart for AMD GPUs is termed wavefront. When there exists branch divergence within a warp, the threads would consume the instructions in both branch paths and each thread only reserves the processed result of the path where the branch condition is hit. Turning back to the CFG, the basic blocks within different branches are consecutively visited as if they are sequentially processed.

We generate the execution trace in granularity of warps. Therefore for the case when the branch condition is thread-ID-dependent, the branch miss information is transformed and associated with the warp ID, given the global and local work size configurations. The basic block is represented as the data structure shown in Listing 1.

```
struct BasicBlockInfo {
    string BBName; // name of current BB
    list<int> branchMissWarpID; // IDs of the warps that trigger branch miss
    // branch miss info at different loop iteration
    // string: name of the basic block that triggers the branch miss
    // int: the exact iteration number for basic block #string when branch miss
    map<string, int> branchMissLoopConfig;
    int loopDepth; // greater than 1 when current BB is in a loop
    string loopBoundExpr; // the loop bound expression
    // BBs of which the loop bounds determine current BB's loop bound
    vector<string> associatedBBs;
    string precedBB; // preceding BB closest to current BB at upper loop level
    vector<int> bounds; // integer values of loop bounds at each loop level
    vector<int> unvisitedCount; // store the visited counters at each loop level
    bool isVisited; // true if current BB is visited over at each loop level
};
list<BasicBlockInfo> BBInfoList; // list of data description for BBs in CFG
```

Listing 1: Sample code of the basic block data description.

The information about the branch miss due to the warp divergence and loop iterations is respectively stored in the `branchMissWarpID` and `branchMissLoopConfig` fields. The `loopDepth` field indicates the loop depth of the basic block. Particularly, this value is set to 1 if the basic block is not in a loop. The analyzed loop bound result is stored in the `loopBoundExpr` field. As this expression only indicates the loop bound of the basic block at its current loop level, the actual loop bound values at each loop level are calculated each time this basic block is visited and these values are stored in the `bounds` field. If the loop bound of the basic block is dependent on other basic blocks, these associated basic blocks are stored as well (the `associatedBBs` field). The preceding basic block that is closest to the current basic block but lies at the upper loop level is stored in the `precedBB` field so as to maintain the nest loop chain. During the execution trace generation, the visited counters of the basic block (the `unvisitedCount` field) are recorded to indicate the visited status of the basic block, i.e., at which loop level and with how many times the current basic block is already visited. The `isVisited` boolean is set to `TRUE` only if the basic block is visited over at each loop level with the number of times equal to the actual loop bound. Finally, the data descriptions of all the basic blocks in the CFG are stored in a list `BBInfoList`.

Given the kernel CFG $G = (\mathbb{V}, \mathbb{E}, \texttt{B}, \texttt{E})$, where $\mathbb{V}$ is the set of basic block nodes, $\mathbb{E}$ is the set of basic block connections, B is the entry node and E is the exit node, the kernel execution trace is generated via a loop-based bidirectional branch search shown in Algorithm 1. The CFG is first passed to a circular check to spilt the edge set $\mathbb{E}$ into the backedge set $\mathbb{BE}$ and the non-backedge set $\mathbb{NE}$. In this way, the CFG is transformed into a Directed Acyclic Graph (DAG) and the paths between any two nodes can be represented as finite sequences of which all the nodes belong to the non-backedge set $\mathbb{NE}$. By default we have the following assumption:

Denote $\mathbb{V}_{\boldsymbol{c}}$ as a set of nodes that construct a circle $\boldsymbol{c}$ in the CFG, if there exists another circle node set $\mathbb{V}_{\boldsymbol{c'}}$, then formula $(\mathbb{V}_{\boldsymbol{c}} \subset \mathbb{V}_{\boldsymbol{c'}}) \vee (\mathbb{V}_{\boldsymbol{c}} \supset \mathbb{V}_{\boldsymbol{c'}}) \vee (\mathbb{V}_{\boldsymbol{c}} \cap \mathbb{V}_{\boldsymbol{c'}} = \varnothing)$ always holds.

This assumption is reasonable for real-world program because a node in a loop can only be reached from the nodes in its surrounding loops but can never reach the nodes in another loop that is beyond all of the outer loop layers of the original loop. The above assumption ensures that no backedge would wander among different circles in the CFG.

We perform a loop-based bidirectional branch search of the CFG to generate the kernel execution trace. As shown in Algorithm 1, the execution trace starts from the entry node B and terminates when the exit node E is visited. A node stack ST is used to store the header nodes of multiple branch paths. An array M is used to store the mask values for each node in the candidate trace $\mathbb{T}$. The mask value is set to 0 when the node to be appended to $\mathbb{T}$ is a branch miss node. For each candidate node τ to be appended to $\mathbb{T}$, a function `updateExecTrace()` is invoked to update the visited counters of τ and another function `getTraceSuccNode()` is used to obtain the

```
Algorithm 2: updateExecTrace($\tau$, $\mathbb{T}$, BBInfoList, wid, $M$)
Input: Candidate Node $\tau$, Candidate Trace $\mathbb{T}$, Basic Block Data
       Description List BBInfoList, Warp ID wid, Mask Array
       $M$
1  $\tau$.bounds ← calcLoopBound($\tau$, BBInfoList) ▷ update loop bounds
2  loopLevelVisitedCount ← 0, unvisitedLoopLevel ← 0
3  isBranchMissWarp ← FALSE, isBranchMissLoop ← FALSE
4  for $i \leftarrow 0$ to $\tau$.loopDepth do
5      if $\tau$.unvisitedCount⟨$i$⟩ = 0 then        ▷ i-th loop level is visited
6          loopLevelVisitedCount ← loopLevelVisitedCount+1
7      else      ▷ currently the trace iterates exactly at the i-th level of the loop
8          unvisitedLoopLevel ← $i$
9          break
10 if loopLevelVisitedCount ≠ $\tau$.loopDepth then
11     for $j \leftarrow 0$ to unvisitedLoopLevel do              ▷ reset loop bounds
12         $\tau$.unvisitedCount⟨$j$⟩ ← $\tau$.bounds⟨$j$⟩
13     $\tau$.unvisitedCount⟨0⟩ ← $\tau$.unvisitedCount⟨0⟩ − 1
14 else ▷ $\tau$ is visited over when the visited-loop-level count equals the loop depth
15     $\tau$.isVisited ← TRUE
16 isBranchMissWarp ← checkBranchMissWarp($\tau$, wid)
17 isBranchMissLoop ← checkBranchMissLoop($\tau$, BBInfoList)
   // set the mask value to 0 when $\tau$ is a branch miss node, otherwise set it to 1
18 $M$.add(¬ isBranchMissWarp ∧ ¬ isBranchMissLoop)
19 $\mathbb{T} \leftarrow \mathbb{T} + \{\tau\}$
```

successor node of τ to be appended to $\mathbb{T}$. Finally, the branch miss nodes are removed from $\mathbb{T}$, based on the mask array M, to generate the kernel execution trace **T**.

The implementation of function `updateExecTrace()` is shown in Algorithm 2. First, the loop bounds of the candidate node τ can be determined because these values are related to the loop bound expression (τ`.loopBoundExpr`) and the current loop iterations and loop bounds of the associated basic blocks (τ`.associatedBBs`), and all these information can be calculated before visiting τ at its current loop level (Line 1 in Algorithm 2). Subsequently, the visited counters of τ are checked to determine at which loop level the node τ is visited (Line 4−7 in Algorithm 2). Each time the unvisited count value at the innermost loop level is decreased by 1 (Line 13 in Algorithm 2). The update of the visited counters is implemented via a decrement operation with borrowing, i.e., each time the unvisited count value at loop level λ is reduced to zero, this value is reset to the loop bound at loop level λ and the unvisited count at loop level $(\lambda+1)$ is decreased by 1 (Line 11−11 in Algorithm 2). If the unvisited count values of τ at all loop levels are zero, then this node is labeled as visited (Line 15 in Algorithm 2). At last, the branch miss information is used to determine whether τ is a branch miss mode. The corresponding mask value is written to the mask array M and node τ is appended to the candidate trace $\mathbb{T}$ (Line 16−19 in Algorithm 2).

Algorithm 3 gives the detailed implementation of the function `getTraceSuccNode()`. To find the successor node of τ to be appended to $\mathbb{T}$, the backedge set $\mathbb{BE}$ is first searched to get the destination node (element in $\mathbb{D}_{\mathbb{BE}}$) of the backedge whose source node is τ (Line 2−2 in Algorithm 3). The candidate backedge nodes (elements in $\mathbb{D}'_{\mathbb{BE}}$) are chosen from the nodes in $\mathbb{D}_{\mathbb{BE}}$ of which the unvisited count value at the innermost loop level equals neither zero nor the loop bound value (Line 4−4 in Algorithm 3). The successor

```
Algorithm 3: getTraceSuccNode($\tau$, B, BBInfoList, ST, $\mathbb{T}$,
             $\mathbb{BE}$, $\mathbb{NE}$)
Input: Current Trace Tail Node $\tau$, CFG Entry Node B, Basic Block
       Data Description List BBInfoList, Node Stack ST,
       Candidate Trace $\mathbb{T}$, Backedge Set $\mathbb{BE}$, Non-backedge Set $\mathbb{NE}$
Output: Candidate Trace Successor Node $\tau$ (overwritten)
1  $\mathbb{D}_{\mathbb{BE}} \leftarrow \varnothing$, $\mathbb{D}'_{\mathbb{BE}} \leftarrow \varnothing$, $\mathbb{D}_{\mathbb{NE}} \leftarrow \varnothing$, $\mathbb{D}'_{\mathbb{NE}} \leftarrow \varnothing$, $\mathbb{S}_{\mathbb{NE}} \leftarrow \varnothing$
   // first try to find a candidate successor node from the backedges
2  if $\exists$ be $\in \mathbb{BE}$, $\tau$ = be.srcNode then
3      $\mathbb{D}_{\mathbb{BE}}$ = {be.destNode | be $\in \mathbb{BE}$, $\tau$ = be.srcNode}
4      foreach $d_{be} \in \mathbb{D}_{\mathbb{BE}}$ do ▷ get candidate nodes that are not visited over
5          if $d_{be}$.unvisitedCount⟨0⟩ % $d_{be}$.bounds⟨0⟩ ≠ 0 then
6              $\mathbb{D}'_{\mathbb{BE}} \leftarrow \mathbb{D}'_{\mathbb{BE}} + \{d_{be}\}$
7      if $\mathbb{D}'_{\mathbb{BE}} \neq \varnothing$ then
8          if $\tau \in \mathbb{D}'_{\mathbb{BE}}$ then          ▷ there is a backedge from $\tau$ to itself
9              return $\tau$          ▷ $\tau$ is not visited over at its current loop level
10         else
11             foreach $d'_{be} \in \mathbb{D}'_{\mathbb{BE}}$ do
12                 $\mathbb{P}_{\mathbb{B}}$ ← getNodesInPath($d'_{be}$, $\tau$)
13             $\mathbb{I}_{\mathbb{BE}} \leftarrow \mathbb{D}'_{\mathbb{BE}} \cap \mathbb{P}_{\mathbb{B}}$
14             if $\mathbb{I}_{\mathbb{BE}} \neq \varnothing$ then
15                 return $\mathbb{I}_{\mathbb{BE}}\langle 0\rangle$          ▷ return the closest-to-$\tau$ node
   // backedge search fails, try to find the successor node from the non-backedges
16 else if $\exists$ ne $\in \mathbb{NE}$, $\tau$ = ne.srcNode then
17     $\mathbb{D}_{\mathbb{NE}}$ = {ne.destNode | ne $\in \mathbb{NE}$, $\tau$ = ne.srcNode}
18     foreach $d_{ne} \in \mathbb{D}_{\mathbb{NE}}$ do     ▷ get the closest-to-$\tau$ non-backedge nodes
19         $\mathbb{P}_{\mathbb{N}}$ ← getNodesInPath($\tau$, $d_{ne}$)
20         if $\mathbb{D}_{\mathbb{NE}} \cap \mathbb{P}_{\mathbb{N}} = \varnothing$ then
21             $\mathbb{D}'_{\mathbb{NE}} \leftarrow \mathbb{D}'_{\mathbb{NE}} + \{d_{ne}\}$
22     if $\mathbb{D}'_{\mathbb{NE}} \neq \varnothing$ then
23         foreach $d'_{ne} \in \mathbb{D}'_{\mathbb{NE}}$ do          ▷ get nodes in other backedges
24             if $\exists$ $be^n \in \mathbb{BE}$, $d'_{ne}$ = $be^n$.srcNode then
25                 $\mathbb{S}_{\mathbb{NE}} \leftarrow \mathbb{S}_{\mathbb{NE}} + \{d'_{ne}\}$
26         $s_{ne}$ = ($\mathbb{S}_{\mathbb{NE}} \neq \varnothing$) ? $\mathbb{S}_{\mathbb{NE}}\langle 0\rangle$ : $\mathbb{D}'_{\mathbb{NE}}\langle 0\rangle$      ▷ candidate successor
27         if ST ≠ ∅ then
28             $\mathbb{P}_{\mathbb{S}}$ ← getNodesInPath(B, $s_{ne}$)
29             if ST⟨topElement⟩ $\in \mathbb{P}_{\mathbb{S}}$ then
30                 $\tau$ ← ST⟨topElement⟩, ST.pop()
31             else          ▷ the stack top node denotes another branch path
32                 $\tau \leftarrow s_{ne}$          ▷ but the current path is not visited over
33             return $\tau$
34         else          ▷ the current path is the last path of the current branch
35             $\mathbb{D}'_{\mathbb{NE}} \leftarrow \mathbb{D}'_{\mathbb{NE}} - \{s_{ne}\}$  ▷ return the candidate successor node
               foreach $d''_{ne} \in \mathbb{D}'_{\mathbb{NE}}$ do ▷ store the remaining header nodes
36                 ST.push($d''_{ne}$)
37             return $s_{ne}$
   // all edges starting from $\tau$ are visited, get the successor from the node stack
38 else
39     $\tau$ ← ST⟨topElement⟩, ST.pop()
40     return $\tau$
```

node of τ to be appended to $\mathbb{T}$ is either itself if τ is in $\mathbb{D}'_{\mathbb{BE}}$ or the closest-to-τ node in the intersection set of $\mathbb{D}'_{\mathbb{BE}}$ and the path node set $\mathbb{P}_{\mathbb{B}}$ in which each node denotes a reachable path to τ (Line 8−10 in Algorithm 3).

If there exists no backedge that starts from τ, or all the backedges starting from τ are visited N times where N is the loop bound in the innermost level, the non-backedge set $\mathbb{NE}$ is searched to obtain the closest-to-τ non-backedge destination node set $\mathbb{D}'_{\mathbb{NE}}$ (Line 16−16 in Algorithm 3). The first node in $\mathbb{D}'_{\mathbb{NE}}$ is chosen as a candidate successor node s_{ne} if none of the nodes in $\mathbb{D}'_{\mathbb{NE}}$ is a source node of a backedge, otherwise this source node becomes s_{ne} (Line

26 in Algorithm 3). If node stack ST is not empty and the stack top node $\mathtt{ST}\langle\mathtt{topElement}\rangle$ lies between a reachable path from the entry node B to s_{ne}, then the successor node of τ to be appended to $\mathbb{T}$ is $\mathtt{ST}\langle\mathtt{topElement}\rangle$, otherwise the successor node to be appended to $\mathbb{T}$ is s_{ne} (Line 27−27 in Algorithm 3). If ST is empty, then s_{ne} is the successor node of τ to be appended to $\mathbb{T}$ and the remaining nodes in $\mathbb{D}'_{\mathbb{NE}}$ are pushed into ST (Line 35−34 in Algorithm 3).

If all the edges starting from τ are visited, then the stack top node $\mathtt{ST}\langle\mathtt{topElement}\rangle$ is popped as successor node of τ to be appended to $\mathbb{T}$ (Line 39−40 in Algorithm 3).

3) Cache behavior analysis: As modern GPUs have rather complex memory hierarchy that comprises caches, we first use micro-benchmarks to obtain the cache hit and miss latencies of the local, constant, and global memory accesses. As the local memory in OpenCL is mapped to the GPU shared memory, we notice that the local memory access has no caching issue and therefore does not differentiate the cache hit/miss access, which is also observed and demonstrated by the micro-benchmarking results. When handling the constant and the global memory accesses, the SMs first try to fetch the data in the constant or L2 data cache and if cache miss occurs, the data are fetched again from the off-chip DRAM. To model this caching behavior, we dissect the constant data cache and the L2 cache with micro-benchmarks [21] [22] to obtain the detailed cache configurations, such as the cache size, the cache line size, and the cache associativity. In OpenCL, the observed constant memory size is 64KB and the DRAM size is obtained from the official documents. The L2 cache size is obtained from the CUDA built-in querying commands. We assume that all the caches use the least recently used (LRU) replacement policy.

For each memory access, i.e. the `load` or `store` IR instruction in the execution trace, we obtain the memory referencing address and analyze the number of memory transactions that a warp would perform for this memory instruction, since the threads in a warp often coalesce the data fetch if the memory addresses for the threads are contiguous. As we do not execute the kernel on the real platform, we construct a virtual addressing space of the constant data cache and the L2 cache, and then assign the specific addresses to the constant and global variables according to their data size. In this way, the cache behavior is analyzed using the reuse distance theory and the cache hit/miss for each memory transaction is estimated given the cache configuration [23].

4) Discussion: Limitation As we do not use profiling or measurement results of the target kernel, the execution behavior of irregular kernels cannot be exactly determined by the static analysis. Consequently the loop bound analysis and the warp-based branch analysis produce slightly pessimistic results when the values of the loop trip count and the branch condition rely on the values of the program runtime parameters. However, the major part of the applications that can potentially benefit from GPU acceleration exhibit relatively regular shapes, i.e., the loop trip count is rather stable and the number of branches is minimized by the program developer as well. With regards to the kernels with data-dependent divergence, because the static analysis module can still extract the branch condition and loop iteration variables of the control statements, the dynamic execution flow can also be determined if all the input data are known in advance. However this needs the step-by-step simulation of the program execution, which may incur much more time consumption. This is one aspect of future work.

Scalability analysis The proposed performance analysis framework in this paper targets OpenCL kernels and therefore it can be extended to any platform that supports OpenCL applications. For other parallel languages such as CUDA, since our framework takes LLVM bitcode files as input, CUDA kernels can also be analyzed if either the LLVM bitcode file of the kernel can be obtained or the CUDA kernels can be transformed into the OpenCL counterparts.

IV. Trace-based simulation

The execution trace **T** generated from the source-level analysis is warp-ID-dependent and during the simulation each warp consumes its corresponding trace. To estimate the kernel execution time with given program input and the global and local work size configurations, we construct an IR instruction pipeline and then simulate the trace on the pipeline in granularity of a round of active work groups.

A. IR instruction pipeline

1) Determining the number of active work groups: Given a kernel with NDRange configuration as global work size S_{global} and local work size S_{local}. Each work item consumes N_{reg} on-chip registers (private memory) and N_{sm} bytes shared memory (local memory). The number of active work groups N_{awg} per Streaming Multiprocessor (SM) is subject to three constraints: the architectural limit, the register limit, and the shared memory limit. The architectural limit of the allocatable work groups is

$$N_{lim\_wg\_arch} = min(B_{wg\_SM}, \lfloor \frac{B_{warp\_SM}}{N_{warp\_per\_wg}} \rfloor) \quad (1)$$

$$N_{warp\_per\_wg} = \lceil \frac{S_{local}}{T_{warp}} \rceil \quad (2)$$

where $N_{warp\_per\_wg}$ is the number of warps per work group, T_{warp} is the number of threads per warp, B_{wg_SM} and B_{warp_SM} is the maximum allocatable work groups and warps per SM, respectively. The number of total on-chip registers limits the maximum concurrent work group as

$$N_{lim\_wg\_reg} = \begin{cases} 0, & N_{reg} > B_{reg\_wi} \\ \lfloor \frac{N_{lim\_warp\_reg}}{N_{warp\_per\_wg}} \rfloor \times \lfloor \frac{B_{reg\_SM}}{B_{reg\_wg}} \rfloor, & otherwise \end{cases} \quad (3)$$

$$N_{lim\_warp\_reg} = floor(\frac{B_{reg\_wg}}{ceil(N_{reg} \times T_{warp}, G_{reg})}, G_{warp}) \quad (4)$$

where B_{reg_wi}, B_{reg_SM}, and B_{reg_wg} are the maximum allocatable registers per work item, SM, and work group,

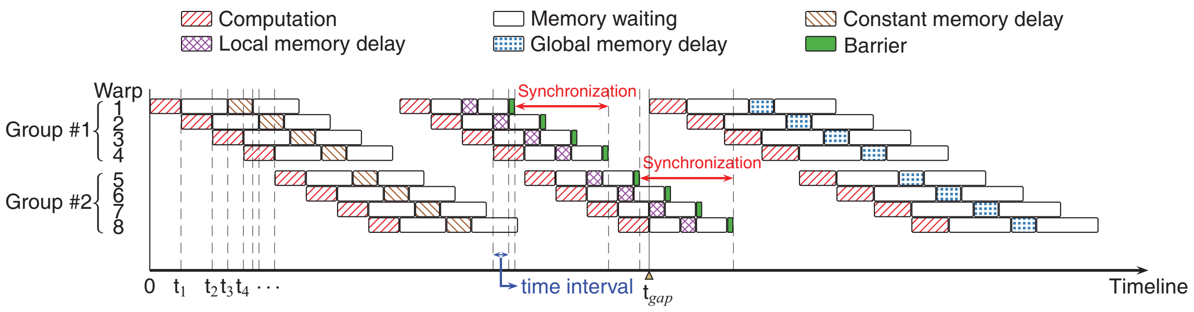


Figure 2: Simulation of a sample execution trace on the warp pipeline.

respectively. G_{reg} and G_{warp} are the minimum allocation unit of register and warp, respectively. $N_{lim\_warp\_reg}$ is the maximum number of potentially allocatable active warps subject to limited on-chip registers. $ceil(x,y)$ and $floor(x,y)$ are functions used to round the value x up and down to the nearest multiple of y, respectively. The number of active work groups due to shared memory limit is calculated as

$$N_{lim\_wg\_sm} = \begin{cases} 0, & N_{sm\_alloc} > B_{sm\_wg} \\ \lfloor \frac{B_{sm\_SM}}{N_{sm\_alloc}} \rfloor, & otherwise \end{cases} \tag{5}$$

$$N_{sm\_alloc} = ceil(N_{sm}, G_{sm}) \tag{6}$$

where B_{sm_wg} and B_{sm_SM} are the maximum allocatable shared memory size per work group and SM, respectively. N_{sm_alloc} is the actual allocated shared memory size per work group and G_{sm} is the minimal shared memory allocation size.

With Equation (1), (3) and (5), the number of active work groups for a kernel is therefore determined as

$$N_{awg} = min(N_{lim\_wg\_arch}, N_{lim\_wg\_reg}, N_{lim\_wg\_sm}) \tag{7}$$

2) Determining the latencies of the arithmetic and memory access operations: The execution trace consists of the arithmetic and memory access operations to be executed on the target GPU. To obtain the latencies of these operations, we use a set of OpenCL micro-benchmarks to measure the arithmetical and memory throughout of the target GPU [24]. We consider the basic arithmetic operations listed in Table I and the latencies of memory access from the OpenCL local, constant, and global memory. The private memory access is essentially on-chip register read/write and this memory access is deemed arithmetic operation since the pre-allocated registers are excluded by individual work item and therefore accessing them incurs no contention latency. The profiling of basic arithmetic operations is conducted over a set of computation-intensive kernels which repeatedly execute the desired operations for millions of times. To prevent the compiler optimization, the source and destination operands are exchanged after each time the operation is completed. By fine tuning the local and global work size of each kernel, the number of active warps per SM is thereupon dynamically regulated so as to obtain the corresponding execution latencies ranging from the minimal to the maximal attainable number of active warps.

With regards to the memory access, we use pointer chasing to generate continuous data access to a large array filled in the respective memory space. To measure the cache hit and miss latencies, the pointer chasing stride offset is set to 1 and the cache line size, respectively. During the simulation, the memory latency is scaled with a factor equaling the ratio of the maximal to the actual number of active warps, since all the active warps share the memory bandwidth equally. The profiled results of the memory access characterize the average time period that starts from the memory instruction issue stage to the final data acquisition stage. We term this whole time period as the memory access "latency" and this time cost is differentiated from the time period when the data is actually read/written by the hardware control circuit, which is called memory access "delay". We assume memory access delay is fixed while memory access latency varies depending on whether the access is a cache hit or miss.

B. Calculating the trace simulation time

Given the kernel execution trace $\mathbf{T}$, the latencies of the arithmetic and memory access operations `LAT`, and the cache miss information `cacheMissInfo` in the trace, we develop a lightweight simulator to manoeuvre a dummy execution of the kernel with a round of active work groups N_{awg}. A sample simulation of this active work groups is conducted on the IR instruction pipeline and the time consumption can be denoted as $T^{spec(\texttt{LAT},\texttt{cacheMissInfo})}_{pipeline(N_{awg},\mathbf{T})}$. The estimated execution time of the kernel run is

$$T_{kernel} = T^{spec(\texttt{LAT},\texttt{cacheMissInfo})}_{pipeline(N_{awg},\mathbf{T})} \times \lceil \frac{S_{global}}{S_{local} \times N_{SM}} \rceil \times \frac{1}{N_{awg}} \tag{8}$$

where N_{SM} is the total number of SMs on the target GPU.

The trace simulation is implemented with a group of active warps continually consuming the arithmetic and memory access operations in presence of the shared resource and cache contention. For each memory access, we assume memory read/write delay is constant while the waiting period of servicing memory read/write varies depending on whether the memory access is cache hit or miss. We model the latency of memory read/write as three parts: the pre-waiting latency, the read/write delay and the post-waiting latency, of

Table I: List of profiled arithmetic operation types.

Data type	Operations
int/uint	add, sub, mul, div, rem, mad, shl, shr
float/double	add, sub, mul, div, mad
float/double	sin, cos, tan, exp, log, sqr, sqrt

Table II: Summary of the parameters used in our performance estimation framework.

No.	Parameter	Definition	Obtained
1	S_{global}	Number of global work size	Program configuration
2	S_{local}	Number of local work size	Program configuration
3	N_{reg}	Number of registers used per work item	Kernel compilation
4	N_{sm}	Bytes of shared memory used per work item	Kernel compilation
5	B_{wg_SM}	Maximum allocatable work groups per SM	Hardware specification
6	B_{warp_SM}	Maximum allocatable warps per SM	Hardware specification
7	B_{reg_wi}	Number of maximum allocatable registers per work item	Hardware specification
8	B_{reg_SM}	Number of maximum allocatable registers per SM	Hardware specification
9	B_{reg_wg}	Number of maximum allocatable registers per work group	Hardware specification
10	B_{sm_wg}	Bytes of maximum allocatable shared memory per work group	Hardware specification
11	B_{sm_SM}	Bytes of maximum allocatable shared memory per SM	Hardware specification
12	G_{sm}	Number of minimum allocation bytes of shared memory	Hardware specification
13	G_{reg}	Number of minimum allocation unit of registers	Hardware specification
14	G_{warp}	Number of minimum allocation unit of warps	Hardware specification
15	$FREQ_{core}$	Clock frequency of the thread core on target GPU	Hardware specification
16	N_{SM}	Number of SMs on target GPU	Hardware specification
17	T_{warp}	Number of thread cores per warp	Hardware specification
18	N_{awg}	Number of active work groups	Equation 7
19	`LAT`	Latencies of arithmetic and memory access operations	Micro-benchmarking
20	`cacheMissInfo`	Cache hit/miss information about the memory access	Cache behavior analysis
21	$\mathbf{T}$	Kernel execution trace	Algorithm 1
22	$T^{spec(\texttt{LAT},\texttt{cacheMissInfo})}_{pipeline(N_{awg},\mathbf{T})}$	Estimated kernel execution time with a round of active work groups	Simulation
23	T_{kernel}	Estimated total kernel execution time	Equation 8

which the sum is the profiled cache hit or miss latency.

For better illustration, Figure 2 gives an example to illustrate how an execution trace is fed into the warp pipeline. The sample trace is defined as (*comp, constMemAccess, comp, localMemAccess, barrier, comp, globalMemAccess*). The number of active work groups is 2 and each work group consists of 4 warps. Each time before a warp consumes a new operation in the trace, it will first check whether the required contention resource is idle. If so it would lock the resource and notify a value denoting the latency of consuming the current operation, otherwise it would notify a value denoting the time needed to wait until the resource is released. If the warp hits a barrier for synchronization, it will notify value 0 and wait for the other warps in the same work group to arrive at this barrier. A global timer starts at time point 0 and increases by a unit of time interval (indicated by the time point of $t_1, t_2, t_3, \ldots$ on the Timeline-axis in Figure 2) when all the active warps have notified a time value. During every time interval, the timer checks the notification time of each warp and chooses the minimum positive time value as the incremental time interval. Once all the active warps finish their own traces, the global timer gives the total time of consuming the execution trace.

C. Discussion and summary

As observed in Figure 2, the execution time of the sample trace is computation-bound and the synchronization latency is hidden by the computation pipeline. However, if there exist more memory access operations before the barrier, there would be a gap between the 2-nd and 3-rd computation component (indicated by time point t_{gap} in Figure 2) and in this case the synchronization latency would contribute to the final execution time. Consequently, analytical performance estimation methods are normally subject to kernel variances because the order that the computation and memory components appear in the execution trace is inconstant and unpredictable, which has a tremendous impact on calculating the consuming latency of the instruction pipeline.

Table II summarizes the parameters used in our proposed framework. As shown, our method requires neither the pre-execution of the whole or a portion of the target kernel nor the profiled results of the hardware performance counter metrics. The used information are the program configuration parameters, kernel compilation report, and the hardware specifications. The micro-benchmarking metrics are obtained by calibrating the target GPU once and these data can be reused for the performance prediction of all the kernels running on this platform. During the simulation, each kernel takes the same kernel compilation results and the same group of execution traces as inputs. For each specific run, only the corresponding global and local work size configurations are fed to the simulator to obtain the estimated results. Moreover, only a round of active work groups is actually fed to the pipeline and therefore the simulation time cost is small.

Overall speaking, compared with traditional architectural simulation methods [25], the proposed framework requires less input information and can give faster estimation outcomes. Our framework does not require the instruction trace representatives generated from the kernel runs, which is subject to specific workloads and may incur substantial effort when the input parameters vary a lot.

V. Experiments and discussion

A. Experimental setup

We use four COTS GPUs to evaluate our performance estimation framework and the detailed information is shown in

Table III: Hardware specification of the test GPUs.

Name	Architecture	SMs/Cores	Clock freq.(MHz)
Quadro K600	Kepler GK107	1/192	876
GeForce GTX645	Kepler GK106	3/384	824
Quadro K620	Maxwell GM107	3/384	1058
GeForce 940M	Maxwell GM108	3/384	1072

Table IV: Accuracy and simulation time consumption of testing our framework on the Rodinia [17] benchmark.

Benchmark name	**Kernel name**	**Number of total design configurations**	**Average trace length**	**MAPE (%)**				**Time per run (ms)**
				Quadro K600	**GeForce GTX645**	**Quadro K620**	**GeForce 940M**	
backprop	bpnn_adjust_weights	11,450	41	24.24	24.34	22.16	22.12	23.03
	bpnn_layerforward	11,450	74	19.38	27.05	21.14	26.55	40.08
bfs	BFS_1	14,028	79	11.55	7.969	10.16	20.47	60.09
	BFS_2	14,028	7	14.43	20.24	9.879	11.73	10.40
b+tree	findK	42,000	100	35.68	31.12	8.941	12.86	72.93
	findRangeK	42,000	163	40.63	39.80	13.42	13.42	119.66
cfd	compute_flux	3,072	616	15.32	19.81	9.077	14.41	77.55
	compute_step_factor	3,072	33	12.83	27.76	43.04	**3.315**	14.01
	initialize_variables	3,072	18	11.89	9.149	29.65	7.902	15.38
	memset	12	2	8.085	25.80	6.803	18.83	7.081
	time_step	3,072	31	5.191	18.38	18.04	16.20	23.78
hotspot	hotspot	1,024	22,093	15.36	14.21	**4.325**	9.389	4,130.09
kmeans	kmeans_c	40,000	2,338	12.95	22.99	19.06	20.37	824.60
	kmeans_swap	40,000	533	10.23	19.80	15.76	17.74	219.67
lud	lud_internal	8,267	108	17.41	23.18	8.278	34.18	45.10
nn	nearestNeighbor	66	9	10.89	26.84	7.090	12.38	6.030
nw	nw_kernel1	19,408	1,431	9.228	21.88	9.090	25.86	65.27
	nw_kernel2	19,408	1,431	9.239	24.69	8.551	24.87	63.99
particlefilter	particle_naive	104	52,387	19.59	16.99	13.82	11.93	11,751.93
pathfinder	dynproc	31,025	1,469	**3.716**	**6.560**	14.33	9.737	1,055.97
Average		15,327.9	4,148.15	15.39	21.43	14.63	16.71	**931.33**
				17.04				

Table III. These GPUs are from recent Kepler and Maxwell architectures with different compute capacities so as to demonstrate the robustness of our framework. We test the framework with 20 OpenCL kernels from the Rodinia [17] benchmark. We use the default input from the benchmarks and conduct a design space exploration that results in a total of 306,558 estimation runs. The simulation is performed on a desktop computer with an Intel® Core$^{\text{TM}}$ i7-3770 CPU.

B. Prediction results

1) Accuracy: Table IV presents the experimental results. The third column in Table IV lists the number of total design configurations of each kernel and the fourth column indicates the average number of IR instructions in the execution trace during the simulation. The average MAPE on the four GPUs is 17.04% and on each GPU, the optimal kernel prediction can achieve an average MAPE of less than 7%. Overall, our performance estimation framework is robust and accurate.

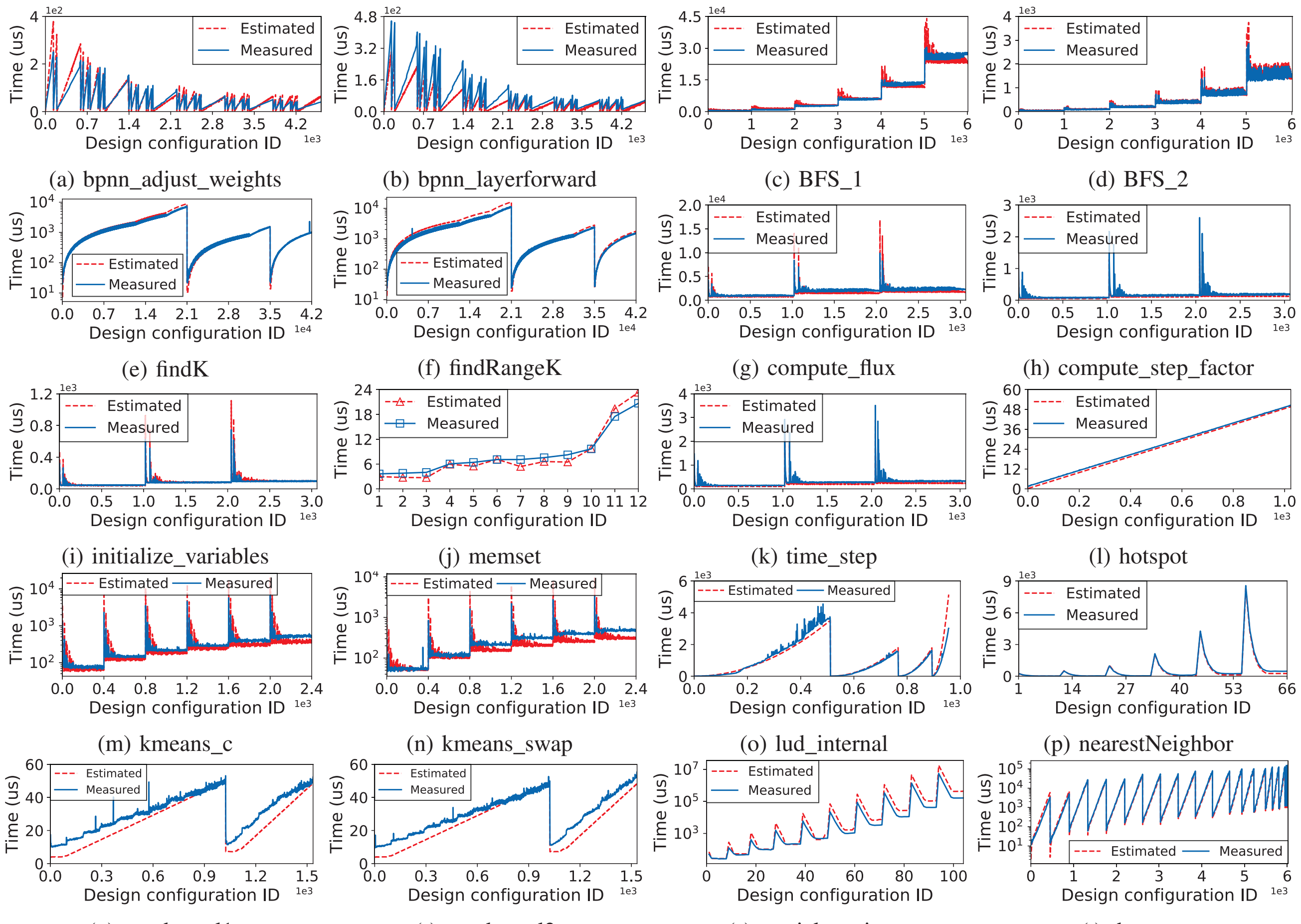
To observe how close our predicted outcome can get to the actual measured results, we plot the result comparison in Figure 3. Due to space limitations, Figure 3 only presents the results of Quadro K620 and the remaining GPUs show similar trends. To clearly show the variation trend of the execution time, for some kernels we only plot partial results in the whole design space because the curves become too dense if the total number of design configurations is too large. The design configuration ID on the *x*-axis represents the number of different program input and local work size settings. The execution time results are sorted in an ascending order with the global and local work size as primary and secondary key, respectively. For some kernels, the program input is also taken as the sorting key. Note that the number of total design configurations is very large and therefore is represented in the scientific notation format, except for kernel `memset`, `nearestNeighbor` and `particle_naive`. The *y*-axes of kernel `findK`, `findRangeK`, `kmeans_c`, `kmeans_swap`, `particle_naive` and `dynproc` are represented in logarithmic scale because the execution time shows several orders of magnitude difference in the absolute value. On the whole view, our predicted results accurately follow the variation trend of the actual execution time across the design space. This reveals that the execution trace and the simulation remarkably reflect the runtime behavior of the kernels, which means that our framework can also help users find the optimal execution even for a vast design space.

As observed in Figure 3, the MAPE turns out higher when the actual execution time is a few microseconds, particularly for kernel `nw_kernel1` and `nw_kernel2` (shown in Figure 3q and 3r). This is because in these cases the kernel overhead dominates the execution time and the predicted time is only a small portion that contributes to the final runtime performance. The kernel overhead includes prerequisite resource allocation, warp scheduling, and kernel launching, etc. The measurement of kernel overhead is infeasible as it is strongly associated with the specific kernel. A possible way is to attach a fixed threshold to the predicted outcome, but again how to set this threshold is pendent.

backprop The MAPEs of this application across four GPUs are quite stable (around 25% in Table IV). The main error source of kernel `bpnn_adjust_weights` is that there are multiple thread-ID-dependent branches and nest branches in the execution flow. Our generated execution trace covers as more branches as possible if the estimated run might step into that branch, thus incurring slight overestimation in some cases (shown in Figure 3a). For kernel `bpnn_layerforward`, the underestimation in Figure 3b comes from barrier synchronization and kernel overhead.

bfs The prediction of this application is better than *backprop*, due to the much less branches. As seen in Figure 3c and 3d, kernel `BFS_1` suffers from larger overestimation than `BFS_2` when the work group size is very small, this is caused by the assumed more cache misses than expected.

b+tree The MAPEs of the kernels in this application



(a) bpnn_adjust_weights (b) bpnn_layerforward (c) BFS_1 (d) BFS_2 (e) findK (f) findRangeK (g) compute_flux (h) compute_step_factor (i) initialize_variables (j) memset (k) time_step (l) hotspot (m) kmeans_c (n) kmeans_swap (o) lud_internal (p) nearestNeighbor (q) nw_kernel1 (r) nw_kernel2 (s) particle_naive (t) dynproc

Figure 3: Comparison of the estimated and measured execution time of the test kernels in Table IV (Quadro K620).

are higher on Kepler than Maxwell GPUs. One possible explanation is that the kernels contain structure data and how these data are organized in memory varies across architectures. Moreover, the multiple runtime-dependent nest branches in the main loop body of both kernels cause workload imbalance and also deteriorate the prediction accuracy.

cfd Estimation of kernel `initialize_variables` shows slightly better accuracy in the variation amplitude (Figure 3i), which is the same case as kernel `memset` (Figure 3j). For the remaining three kernels, the error stems from the variant memory access behavior.

hotspot This application contains rather regular workload distribution across work items and our framework performs the prediction very well, as shown in Figure 3l. The minor underestimation is caused by the kernel overhead, because the execution time of this kernel is less than 60 us.

kmeans Figure 3m and 3n show that predicted outcome of kernel `kmeans_swap` reveals larger fluctuations than `kmeans_c`. We attribute this to the continuous global memory data exchange which incurs irregular memory access.

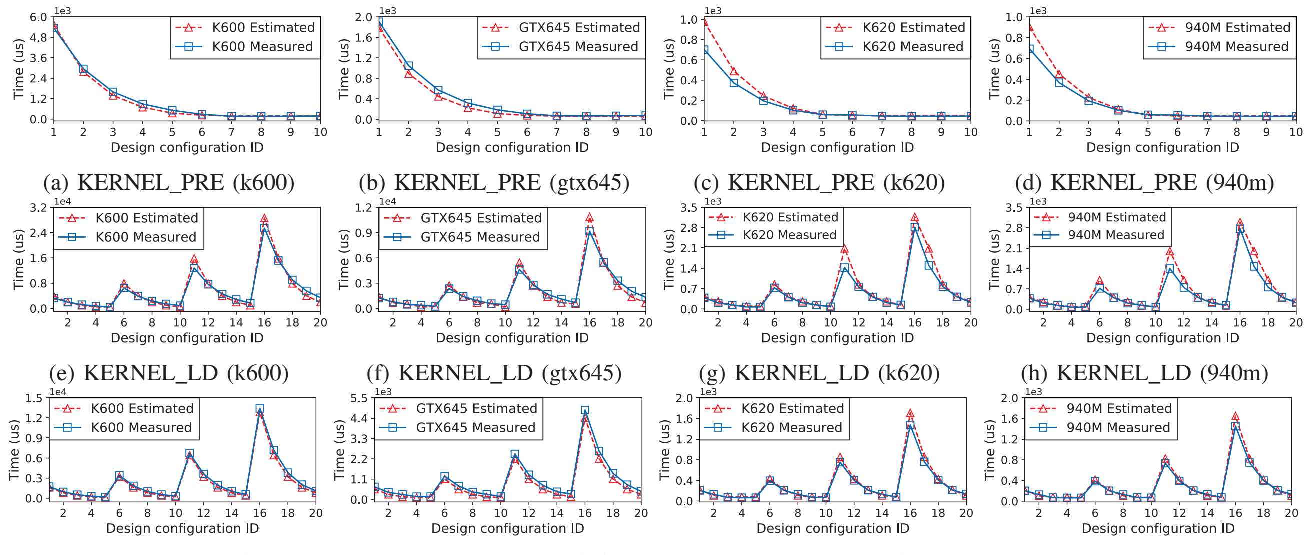
lud & nn These two applications exhibit rather accurate predictions since both kernels have no branch divergence and `lud_internal` only has a loop with fixed bound.

nw Both `nw_kernel1` and `nw_kernel2` have several runtime-dependent branches, which makes the estimation more pessimistic. However, Figure 3q and 3r reveal counter-expectation results. The reason is that kernel overhead also contributes to the MAPE and it is nonnegligible because the total execution time is only a few microseconds. Consequently, kernel overhead compensates for the overestimation and even increases the time consumption for most cases.

particlefilter Our predicted execution time shows overestimation for kernel `particle_naive` in Figure 3s, because there exists runtime-dependent branches in the loop, which constructs the unevenly distributed workload across work items. Our estimation always assumes the longer execution trace for all the warps and therefore is conservative.

pathfinder Similar to `lud` and `nn`, prediction results on this application is rather accurate, as loops are iterated with fixed times and the branches are equally visited by the warps.

To summary, our hybrid framework performs well on the test benchmarks in terms of MAPE. The variation trend of the kernel execution time in the design space is accurately captured by the estimated results. However, the influence of the kernel overhead is significant when the overall execution time is very small, i.e., a few microseconds in our test. In these cases, the dominant factor that contributes to the kernel execution time is not the computation and memory access la-



(a) KERNEL_PRE (k600) (b) KERNEL_PRE (gtx645) (c) KERNEL_PRE (k620) (d) KERNEL_PRE (940m)

(e) KERNEL_LD (k600) (f) KERNEL_LD (gtx645) (g) KERNEL_LD (k620) (h) KERNEL_LD (940m)

(i) KERNEL_PF (k600) (j) KERNEL_PF (gtx645) (k) KERNEL_PF (k620) (l) KERNEL_PF (940m)

Figure 4: Comparison of the estimated and measured results of the lane detection on different GPUs.

tency but the interference from the overhead. Our framework may incur overestimation for irregular workloads, due to the conservative branch divergence analysis. However, note that *bfs* is also an irregular application and our framework can still gives rather good estimation results.

2) Simulation time cost: The last column in Table IV presents the average simulation time of predicting the execution time of each kernel run. As shown, on average our framework can give prediction results within 0.931 second, which is much faster than using a fine-grained simulator [15] [16]. The consumed times of estimating kernel `hotspot` and `particle_naive` are longer than the remaining kernels due to their extremely long execution traces.

We compare the simulation time of our framework with the widely-used GPGPU-Sim [25] and Table V gives the results. As shown, the simulation cost of our method is only a few seconds, while GPGPU-Sim takes time in magnitude of minutes. Our framework achieves an average speedup of 164.39× over GPGPU-Sim, in terms of the simulation time cost, on the test benchmarks.

VI. Case study with lane detection

To demonstrate the effectiveness of the proposed framework, we use a real-world lane detection [26] as test case. The algorithm consists of three steps, namely pre-processing, lane detection, and lane tracking. For each image frame,

Table V: Comparison of the simulation time costs of GPGPU-Sim [25] and our framework.

Benchmark	**Simulation time (ms)**		**Speedup**
	GPGPU-Sim	**Our framework**	
bfs	4,517,000	70.49	64,080.01
hotspot	200,000	4,130.09	48.43
lud	168,000	45.10	3,725.06
nn	3,000	6.030	497.51
nw	1,673,000	129.26	12,942.91
pathfinder	280,000	1,055.97	265.16
Geo. mean	244,433.52	148.69	**164.39**

the pre-processing step extracts the information about the lane markings and then passes the processed image to the next step. Depending on whether the estimated positions of the lane markings in previous frame can still be applied to the current frame, the image is processed either reusing the lane detection step to detect the positions or using particle filter to track the previous positions of the lane markings. The aforementioned three steps are mapped to three kernels and Table VI gives the program configuration of the application during our experiment. For 640×480 input videos, the Region Of Interest (ROI) size of `KERNEL_PRE` is 512×96 and the other two kernels are configured with global work size ranging from 2^{10} to 2^{13}.

We collect the timing information of these kernels for the whole video and then calculate the averaged results per frame. Figure 4 gives the results of the predicted and the measured time. As can be observed, for all the kernels across the different GPUs, the estimations keep the same variation trend with the measured results. The average MAPEs of the three kernels are 15.45%, 19.60% and 17.10%, respectively. The average prediction error for this application is 17.38%.

VII. Related Work

There exist lots of studies targeting performance estimation of applications or benchmarks [27] [28] on CPUs [29] [30]. These approaches can provide reference to the performance analysis of GPUs.

In general, GPU performance estimation techniques can be divided into four categories: analytical, machine learning based, measurement based, and simulation based methods. In the following we briefly summarize these approaches.

Table VI: Configuration of the lane detection kernels.

Kernel name	**Global size**	**Local size**	**No. of designs**
`KERNEL_PRE`	49,152	$\{2^1, 2^2, 2^3, \ldots, 2^{10}\}$	10
`KERNEL_LD`	$2^{10}, 2^{11}, 2^{12}, 2^{13}$	$\{2^1, 2^2, 2^3, 2^4, 2^5\}$	20
`KERNEL_PF`	$2^{10}, 2^{11}, 2^{12}, 2^{13}$	$\{2^1, 2^2, 2^3, 2^4, 2^5\}$	20

Analytical methods first give an abstraction of the workload and hardware, and then use equations [31] to deduce the elapsed time of executing the workload on GPUs. The high-level abstraction metrics are typically thread- and warp-level metrics [32] [33]. Other researchers proposed high level prediction methods [34] [35] based on parallel programming models, such as BSP [36], PRAM [37], and QRQW [38]. Quantitative analysis techniques [39] [40] [41] [42] are also used to abstract the components that contribute to the kernel performance. Most analytical methods require extra dynamic profiling to obtain hardware performance counter metrics and some models are either outdated for new architectures or difficult to use due to the substantial calibration effort. *Machine learning based methods* first construct the training set by sampling program- and platform-related metrics as features and then predict the performance using training models, such as K-nearest clustering [43], regression [7], random forest [9] [44], neural network [8] [45], and so on. Machine learning based methods can estimate the performance with fast response, since the training stage is performed off-line. However, feature sampling of the hardware counter metrics over the huge design space is tedious and the trained model is sensitive to unknown applications. *Measurement based methods* grasp the program behavior by running a portion of the target workload as samples to seek the correlation and interference between individual work groups [46] and then estimate the consumed time when the entire kernel is to be executed. In general, measurement based approaches are universally applicable to different architectures, however the effort to calibrate the model parameters for various applications and platforms is onerous. *Simulation based methods* simulate in details how GPU processes target workloads in cycle level and record the intermediate status of the hardware and software functional modules at runtime. In this way, program behavior and performance can be effectively and accurately sketched [47]. There are some widely-used simulators such as GPGPU-Sim [25], Barra [48] and Ocelot [49]. Recently a RTL-level simulator [50] is announced but few studies are reported.

With regards to GPU simulation acceleration, there exist some research that either choose a portion [51] or perform a pre-characterization [52] of target workloads and then derive the execution time from the simulation results. There are also studies that focus on the generation of GPU benchmarks [53] to reveal GPU's performance spectrum, and modeling of GPU memory systems [54]. These studies are supplementary for GPU performance estimation techniques.

VIII. Conclusion

This paper proposes a hybrid framework to estimate the performance of parallel workloads on GPUs. The high-level source code is analyzed to extract the kernel execution trace, which is used to dynamically mimic the kernel execution behavior to deduce the kernel execution time. Our framework requires no prior knowledge about hardware performance counter metrics or pre-executed measurement results. Experimental results reveal that our framework can accurately grasp the variation trend and predict the execution time with high accuracy and little simulation time cost.

Acknowledgment

This work is supported in part by the China Scholarship Council (CSC) under the Grant Number 201506270152, National Natural Science Foundation of China under the Grant Number 61872393, and National Science Foundation under the Grant Number NSF-CCF-1657333, NSF-CCF-1717754, NSF-CNS-1717984, and NSF-CCF-1750656.

References

[1] A. Nukada and S. Matsuoka, "Auto-tuning 3-d fft library for cuda gpus," in *Proceedings of International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, p. 30, ACM, 2009.

[2] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures.," in *5th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, pp. 111–125, Springer, 2010.

[3] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *ACM SIGPLAN notices*, vol. 45, pp. 115–126, ACM, 2010.

[4] A. Davidson, Y. Zhang, and J. D. Owens, "An auto-tuned method for solving large tridiagonal systems on the gpu," in *IEEE 25th International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 956–965, IEEE, 2011.

[5] C. Nugteren and V. Codreanu, "Cltune: A generic auto-tuner for opencl kernels," in *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 195–202, IEEE, 2015.

[6] T. L. Falch and A. C. Elster, "Machine learning based auto-tuning for enhanced opencl performance portability," in *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pp. 1231–1240, IEEE, 2015.

[7] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pp. 725–737, ACM, 2015.

[8] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 564–576, IEEE, 2015.

[9] K. O'neal, P. Brisk, A. Abousamra, Z. Waters, and E. Shriver, "Gpu performance estimation using software rasterization and machine learning," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 148, 2017.

[10] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort, "The landscape of gpgpu performance modeling tools," *Parallel Computing*, vol. 56, pp. 18–33, 2016.

[11] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. John, H. Jin, and C. Xu, "Accelerating gpgpu architecture simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, pp. 331–332, ACM, 2013.

[12] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, "Tbpoint: Reducing simulation time for large-scale gpgpu kernels," in *IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 437–446, IEEE, 2014.

[13] A. Munshi, "The opencl specification," in *IEEE Hot Chips Symposium (HCS)*, pp. 1–314, IEEE, 2009.

[14] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, p. 75, IEEE Computer Society, 2004.

[15] S. Lee and W. W. Ro, "Parallel gpu architecture simulation framework exploiting work allocation unit parallelism," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 107–117, IEEE, 2013.

[16] G. Malhotra, S. Goel, and S. R. Sarangi, "Gputejas: A parallel simulator for gpu architectures," in *IEEE 21st International Conference on High Performance Computing (HiPC)*, pp. 1–10, IEEE, 2014.

[17] 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 *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, IEEE, 2009.

[18] X. Xie, B. Chen, L. Zou, Y. Liu, W. Le, and X. Li, “Automatic loop summarization via path dependency analysis,” *IEEE Transactions on Software Engineering (TSE)*, no. 1, pp. 1–1, 2017.

[19] M. Sinn and F. Zuleger, “Loopus: A tool for computing loop bounds for c programs.,” in *Proceedings of the Workshop on Invariant Generation (WING)*, pp. 185–186, 2010.

[20] R. A. Van Engelen, “Efficient symbolic analysis for optimizing compilers,” in *International Conference on Compiler Construction (CC)*, pp. 118–132, Springer, 2001.

[21] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying gpu microarchitecture through microbenchmarking,” in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 235–246, IEEE, 2010.

[22] X. Mei and X. Chu, “Dissecting gpu memory hierarchy through microbenchmarking,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 1, pp. 72–86, 2017.

[23] S. Wang, G. Zhong, and T. Mitra, “Cgpredict: Embedded gpu performance estimation from single-threaded applications,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 146, 2017.

[24] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, “Automatic opencl device characterization: guiding optimized kernel design,” in *17th International European Conference on Parallel Processing (Euro-Par)*, pp. 438–452, Springer, 2011.

[25] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing cuda workloads using a detailed gpu simulator,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163–174, IEEE, 2009.

[26] K. Huang, B. Hu, L. Chen, A. Knoll, and Z. Wang, “Adas on cots with opencl: a case study with lane detection,” *IEEE Transactions on Computers (TC)*, 2017.

[27] K. Ganesan, J. Jo, and L. K. John, “Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads,” in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 33–44, IEEE, 2010.

[28] R. Panda, X. Zheng, S. Song, J. H. Ryoo, M. LeBeane, A. Gerstlauer, and L. K. John, “Genesys: Automatically generating representative training sets for predictive benchmarking,” in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 116–123, IEEE, 2016.

[29] J. Chen, L. K. John, and D. Kaseridis, “Modeling program resource demand using inherent program characteristics,” in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and modeling of computer systems*, pp. 1–12, ACM, 2011.

[30] X. Zheng, H. Vikalo, S. Song, L. K. John, and A. Gerstlauer, “Sampling-based binary-level cross-platform performance estimation,” in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pp. 1713–1718, European Design and Automation Association, 2017.

[31] J. Lai and A. Seznec, “Break down gpu execution time with an analytical method,” in *Proceedings of the Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pp. 33–39, ACM, 2012.

[32] S. Hong and H. Kim, “An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness,” in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 152–163, ACM, 2009.

[33] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, “An adaptive performance modeling tool for gpu architectures,” in *ACM SIGPLAN Notices*, vol. 45, pp. 105–114, ACM, 2010.

[34] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, “A performance prediction model for the cuda gpgpu platform,” in *IEEE 16th International Conference on High Performance Computing (HiPC)*, pp. 463–472, IEEE, 2009.

[35] M. Amarís, D. Cordeiro, A. Goldman, and R. Y. de Camargo, “A simple bsp-based model to predict execution time in gpu applications,” in *IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 285–294, IEEE, 2015.

[36] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM (CACM)*, vol. 33, no. 8, pp. 103–111, 1990.

[37] S. Fortune and J. Wyllie, “Parallelism in random access machines,” in *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 114–118, ACM, 1978.

[38] P. B. Gibbons, Y. Matias, and V. Ramachandran, “The queue-read queue-write pram model: Accounting for contention in parallel algorithms,” *SIAM Journal on Computing*, pp. 638–648, 1997.

[39] Y. Zhang and J. D. Owens, “A quantitative performance analysis model for gpu architectures,” in *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 382–393, IEEE, 2011.

[40] S. Song, C. Su, B. Rountree, and K. W. Cameron, “A simplified and accurate model of power-performance efficiency on emergent gpu architectures,” in *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 673–686, IEEE, 2013.

[41] Q. Wang and X. Chu, “Gpgpu performance estimation with core and memory frequency scaling,” *arXiv preprint arXiv:1701.05308*, 2017.

[42] K. Zhou, G. Tan, X. Zhang, C. Wang, and N. Sun, “A performance analysis framework for exploiting gpu microarchitectural capability,” in *Proceedings of the International Conference on Supercomputing (ICS)*, p. 15, ACM, 2017.

[43] I. Baldini, S. J. Fink, and E. Altman, “Predicting gpu performance from cpu runs using machine learning,” in *IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 254–261, IEEE, 2014.

[44] Y. Zhang, Y. Hu, B. Li, and L. Peng, “Performance and power analysis of ati gpu: A statistical approach,” in *IEEE 6th International Conference on Networking, Architecture and Storage (NAS)*, pp. 149–158, IEEE, 2011.

[45] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram, “A comparison of gpu execution time prediction using machine learning and analytical modeling,” in *IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pp. 326–333, IEEE, 2016.

[46] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, “A performance model for gpus with caches,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 7, pp. 1800–1813, 2015.

[47] C. Gerum, O. Bringmann, and W. Rosenstiel, “Source level performance simulation of gpu cores,” in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 217–222, EDA Consortium, 2015.

[48] S. Collange, M. Daumas, D. Defour, and D. Parello, “Barra: A parallel functional simulator for gpgpu,” in *18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 351–360, IEEE, 2010.

[49] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, “Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 353–364, ACM, 2010.

[50] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, “Enabling gpgpu low-level hardware explorations with miaow: an open-source rtl implementation of a gpgpu,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 2, p. 21, 2015.

[51] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. K. John, H. Jin, C. Xu, and J. Wu, “Gpgpu-minibench: Accelerating gpgpu micro-architecture simulation,” *IEEE Transactions on Computers (TC)*, vol. 64, no. 11, pp. 3153–3166, 2015.

[52] K. Punniyamurthy, B. Boroujerdian, and A. Gerstlauer, “Gatsim: abstract timing simulation of gpus,” in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pp. 43–48, European Design and Automation Association, 2017.

[53] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John, “Gpgpu benchmark suites: How well do they sample the performance spectrum?,” in *44th International Conference on Parallel Processing (ICPP)*, pp. 320–329, IEEE, 2015.

[54] R. Panda, X. Zheng, J. Wang, A. Gerstlauer, and L. K. John, “Statistical pattern based modeling of gpu memory access streams,” in *Proceedings of the 54th Annual Design Automation Conference (DAC)*, p. 81, ACM, 2017.