Efficient Performance Estimation and Work-Group Size Pruning for OpenCL Kernels on GPUs

Xiebing Wang, Xuehai Qian, Alois Knoll, and Kai Huang

Abstract—Graphic Processing Units (GPUs) play a vital role in state-of-the-art high-performance scientific computing realm and research work towards its performance analysis is crucial but nontrivial. Extant GPU performance models are far from practical use, while fine-grained GPU simulation requires a considerably large time cost. Moreover, massive amounts of designs with various program inputs and parameter settings pose a challenge for efficient performance estimation and tuning of parallel GPU applications. To this end, this article presents a hybrid framework for the efficient performance estimation and work-group size pruning of OpenCL workloads on GPUs. The framework contains a static module used to extract the kernel execution trace from the high-level source code and a dynamical module used to mimic the kernel execution flow to estimate the runtime performance. For the design space pruning, an extra analysis is performed to filter out the redundant work-group sizes with duplicated execution traces and inferior pipelines. The proposed framework does not require any program runs to estimate the performance and find the optimal or near-optimal designs. Experiments on four Commercial Off-The-Shelf (COTS) Nvidia GPUs show that the framework can predict the runtime performance with an average error of 17.04 percent and reduce the program design space by an average of 78.47 percent.

Index Terms—GPU, performance estimation, OpenCL, work-group size, performance tuning

1 Introduction

The sustained explosive growth of big data has necessitated the deployment of hardware accelerators like Graphic Processing Units (GPUs) to consume computational workloads that are prohibitive for conventional CPUs. Over the last decade, GPUs have attracted an overwhelming amount of attention, and utilization of GPUs has dominated state-of-the-art research about image processing, deep learning, and even embedded system design. To fully exploit GPU's computing power, 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 performance estimation and tuning, are used to help seek the optimal execution of the target application from the vast program design space.

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. Second, 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 [1], [2], [3] 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. At last, there still exists possibility to improve the accuracy and usability of state-of-the-art GPU performance models [4]. Although fine-grained GPU simulators could give rather accurate estimations, the extremely large time consumption makes it unsuitable for practical use [5], [6].

Meanwhile, another critical problem for GPU programmers is how to locate the optimal design of target application so as to deliver the best performance. GPU applications generally take very large workloads as input and these workloads are evenly split and further mapped onto the numerous thread cores that handle substantial data manipulations. Given the huge workload size, the search for a proper sub-workload size that yields optimal performance is nontrivial. To address this issue, two main methodologies are adopted in state-of-the-art research. The first technique, called measurement-based performance auto-tuning [7], [8], samples a portion of the kernel executions selected from the entire design

- *X. Wang and A. Knoll are with the Chair of Robotics, Artificial Intelligence and Real-time Systems, Department of Informatics, Technical University of Munich, 85748 Garching, Germany. E-mail: {wangxie, knoll}@in.tum.de.*
- *X. Qian is with the Ming Hsieh Department of Electrical Engineering, Department of Computer Science, University of Southern California, Los Angeles, CA 90089. E-mail: xuehai.qian@usc.edu.*
- *K. Huang is with the Key Laboratory of Machine Intelligence and Advanced Computing, Ministry of Education, and School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510275, P. R. China. E-mail: huangk36@mail.sysu.edu.cn.*

Manuscript received 26 May 2019; revised 3 Dec. 2019; accepted 3 Dec. 2019. Date of publication 9 Dec. 2019; date of current version 20 Jan. 2020. (Corresponding author: Kai Huang.)
Recommended for acceptance by W. Yu.
Digital Object Identifier no. 10.1109/TPDS.2019.2958343

space and then tries to identify the optimal design with the aid of miscellaneous search strategies [7] or machine learning algorithms [9]. The second approach is program abstraction, which first extracts key features from the static source code [10] or dynamic kernel execution results [11], and then defines metrics that are directly or indirectly correlated with the runtime performance. By optimizing these performance metrics, the optimal designs are finally derived.

While the aforementioned methods are effective for finding the optimal or near-optimal designs, there are still some drawbacks. First, performance auto-tuning depends heavily on the execution results of profiling runs on the target GPUs. This can incur substantial efforts, including program deployment, debugging, and profiling time cost, when applying the method to different platforms. Second, although performance metrics can reasonably reflect the kernel runtime behavior, providing a clear explanation of how these metrics can influence the kernel execution time is not straightforward. Finally, static performance metrics are often subject to specific programming language or hardware platform, so extra effort is needed when applying them to different architectures.

In this article, we present a hybrid framework for the performance estimation and design space pruning of parallel applications running on GPUs. The high-level kernel source code is first transformed into LLVM [12] 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 kernel execution time. For the design space pruning, we target the work-group size selection for the OpenCL kernels. The framework performs an extra analysis to filter out redundant designs with duplicated execution traces and inferior pipelines, and then produces the estimated optimal design by searching the work-group size yielding the minimum predicted kernel execution time.

This article presents a substantial extended work of our earlier study in [13]. Based on the previous work, we investigate the effect of work-group size on the runtime performance of OpenCL kernels and extend the performance estimation method to a holistic performance analysis framework which contains: ① a static module that generates the IR-level execution trace to manifest the kernel execution flow, ② a dynamical module that simulates the trace on a warp-based pipeline so as to obtain the predicted execution time, and ③ an extra execution trace analysis module that can efficiently prune the work-group size settings and locate the optimal designs for OpenCL kernels.

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. For the evaluation, we validate our framework on COTS GPUs with various OpenCL kernels from the Rodinia [14] benchmark. Experiments demonstrate that our framework can accurately grasp the variation trend of the kernel execution time in the design space. On average, the proposed framework can give performance estimation with Mean Absolute Percentage Error (MAPE) of 17.04 percent. Moreover, the framework reduces the program design space by 78.47 percent and needs only 24.67 percent of the time taken by the exhaustive simulation search to find the optimal work-group size. The runtime performance of the test OpenCL kernels with the selected designs is on average only 1.132 times slower than that with the truly optimal design.

2 Related Work

Performance Estimation. 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.

Analytical methods first give an abstraction of the workload and hardware, and then use equations [15] to deduce the elapsed time of executing the workload on GPUs. The high-level abstraction metrics are typically thread- and warp-level metrics [16], [17]. Other researchers proposed high level prediction methods [18], [19] based on parallel programming models, such as BSP [20], PRAM [21], and QRQW [22]. Quantitative analysis techniques [23], [24], [25], [26] 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 [27], regression [1], random forest [3], [28], neural network [2], [29], 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 [30] and then estimate the consumed time when the entire kernel is to be executed. 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 [31]. There are some widely-used simulators such as GPGPU-Sim [32], Barra [33], and Ocelot [34]. Recently a RTL-level simulator [35] is announced but few studies are reported.

Performance Tuning. The literature on performance auto-tuning and design space exploration for parallel applications has grown over the last decade. At first, performance auto-tuning research mainly focused on commonly used but computationally demanding algorithms, such as matrix multiplication [36], [37], stencil computation [38], fast Fourier

transform [39], etc. The popular use of GPUs for general-purpose computing has driven the emergence of auto-tuners used for generic algorithms. Measurement-based auto-tuners prune the program design space with sampling executions and then identify the optimal design using various search strategies [7], [8], [9]. Other studies have proposed performance metrics that are used as optimization object function to derive the parameters related to the optimal executions [10], [11]. There are also studies that combine measurement outcomes and extracted program metrics to identify the optimal parameter settings [40].

Apart from these studies, some researchers have also investigated the work-group size tuning of GPU kernels. Onur *et al.* [41] proposed a dynamical scheduling algorithm to moderate the number of active Cooperative Thread Arrays (CTA) running on each GPU core. They demonstrated the effectiveness of the algorithm on a GPU simulator and the CTA moderation is done during the execution. In contrast, our approach is demonstrated on real GPU products and requires no measurement-based performance metrics. Ari *et al.* [42] proposed a framework that combines analytical model, compiler-time tuning, and runtime adaption to identify the optimal occupancy for a given GPU program. Our work is more static since the design pruning is based on the performance model and static IR analysis results.

3 Framework Overview

Fig. 1 gives the overview of the proposed hybrid framework. The kernel source code is first processed by Clang to generate the LLVM bitcode file that contains IR instructions of the target kernel. Meanwhile, the source file is passed to NVCC to obtain kernel compilation information which 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 three modules, i.e., the source-level analysis module, the trace-based simulation module, and the design pruning module.

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 thereupon consuming the execution trace iteratively. A set of micro-benchmarks are used to calibrate 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 kernel execution time.

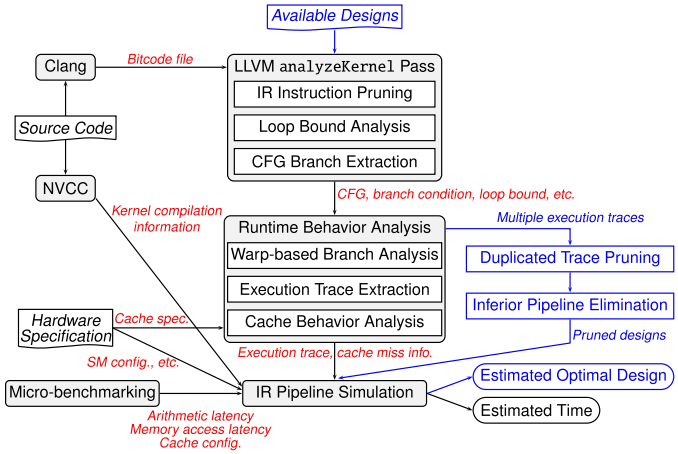


Fig. 1. Overview of the hybrid framework. The blue part indicates the design pruning module.

The design pruning module takes all the available work-group size settings of target OpenCL kernel as input, and then utilizes the source-level analysis module to generate multiple execution traces collected when setting the different work-group sizes. The work-group sizes that yield the same execution traces and the same in-memory layouts for the input kernel arguments, are grouped in one batch, and only the design with the minimum number of rounds of active warps in each batch is chosen as a representative design. The representative designs extracted from different batches are then pruned by filtering out inferior cases in which the lower bound of the best-case execution time is larger than the upper bound of the worst-case execution time yielded by another representative design. The pruned designs are then fed to the simulation module, in which an exhaustive search is performed to estimate the kernel execution time. Finally, the design with the shortest predicted execution time is chosen as the estimated optimal design.

4 Source-Level Analysis

4.1 LLVM `analyzeKernel` Pass

The `analyzeKernel` pass extracts the execution flow of the OpenCL kernels from the IR level and then uses these information to generate the execution trace. This is obtained via the three steps illustrated as follows.

4.1.1 IR Instruction Pruning

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`,[1] the data type conversion instructions `trunc`, `ext`, and so on. Note that, these instructions are only removed from the

1. For the details of LLVM IR instruction description, refer to LLVM Language Reference Manual (https://llvm.org/docs/LangRef.html).

execution trace, but are still used for the later control flow and memory access analysis.

4.1.2 Loop Bound Analysis

We first use Loopus [43] to analyze simple loops and for more complicated loops, the loop bound is first determined by performing an LLVM Scalar Evolution (SE) analysis [44] of the basic blocks in the loop. 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 an 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: $loopBound = \frac{endValue - startValue}{stepSize}$.

4.1.3 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.

4.2 Runtime Behavior Analysis

4.2.1 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 4.1.3, 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 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, as 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.

4.2.2 Execution Trace Generation

We generate the execution trace in granularity of warps. The basic block is represented as the data structure shown in Listing 1. 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. 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. 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.

Given the kernel CFG $G = (\mathbb{V}, \mathbb{E}, \mathtt{B}, \mathtt{E})$, 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 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 $\mathbb{T}$, function `updateExecTrace()` is used to update the visited counters of τ and function `getTraceSuccNode()` is used to obtain the 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 obtain the kernel execution trace **T**.

Listing 1. Sample Code of the Basic Block Data Description

```
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
 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
```

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 information **BBInfoList**, Warp ID wid, Mask array M
Output: Kernel execution trace **T**

1: $\mathbf{T} \leftarrow \emptyset, \mathbb{T} \leftarrow \emptyset, \tau \leftarrow$ B ▷ *Initialize the execution trace with entry node B*
2: updateExecTrace(τ, $\mathbb{T}$, **BBInfoList**, wid, M)
3: ST $\leftarrow \emptyset$ ▷ *Initialize a stack to store header nodes of multiple branch paths*
4: **while** $\tau \neq$ E $\wedge \neg$E.isVisited **do** ▷ *Terminate when E is visited*
5: $\tau \leftarrow$ getTraceSuccNode(τ, B, **BBInfoList**, ST, $\mathbb{T}$, $\mathbb{BE}$, $\mathbb{NE}$)
6: updateExecTrace(τ, $\mathbb{T}$, **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 trace*
9: $\mathbf{T} \leftarrow \mathbf{T} + \{\mathbb{T}.at(i)\}$
10: **return 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 4−9 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 11−12 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 2. updateExecTrace(τ, $\mathbb{T}$, **BBInfoList**, wid, M)

Input: Candidate node τ, Candidate trace $\mathbb{T}$, Basic block information **BBInfoList**, Warp ID wid, Mask array M

1: τ.bounds $\leftarrow$ calcLoopBound(τ, **BBInfoList**)
2: loopLevelVisitedCount $\leftarrow 0$, unvisitedLoopLevel $\leftarrow 0$
3: isBranchMissWarp $\leftarrow$ FALSE, isBranchMissLoop $\leftarrow$ FALSE
4: **for** $i \leftarrow 0$ **to** $\tau.loopDepth$ **do**
5: **if** $\tau.unvisitedCount\langle i\rangle = 0$ **then** ▷ *ith loop level is visited*
6: loopLevelVisitedCount $\leftarrow$ loopLevelVisitedCount+1
7: **else** ▷ *currently the trace iterates exactly at the ith level of the loop*
8: unvisitedLoopLevel $\leftarrow i$
9: **break**
10: **if** *loopLevelVisitedCount* $\neq \tau.loopDepth$ **then**
11: **for** $j \leftarrow 0$ **to** *unvisitedLoopLevel* **do** ▷ *reset loop bounds*
12: τ.unvisitedCount$\langle j\rangle \leftarrow \tau$.bounds$\langle j\rangle$
13: τ.unvisitedCount$\langle 0\rangle \leftarrow \tau$.unvisitedCount$\langle 0\rangle - 1$
14: **else** ▷ *τ is visited over when the visited-loop-level count equals the loop depth*
15: τ.isVisited $\leftarrow$ TRUE
16: isBranchMissWarp $\leftarrow$ checkBranchMissWarp(τ, wid)
17: isBranchMissLoop $\leftarrow$ checkBranchMissLoop(τ, **BBInfoList**)
18: $M.add(\neg$ isBranchMissWarp $\wedge \neg$ isBranchMissLoop)
19: $\mathbb{T} \leftarrow \mathbb{T} + \{\tau\}$

Algorithm 3 gives the detailed implementation of 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 2−15 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 (Lines 4−6 in Algorithm 3). The successor 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 (Lines 8−15 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 16−38 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 ST$\langle$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 ST$\langle$topElement$\rangle$, otherwise the successor node to be appended to $\mathbb{T}$ is s_{ne} (Line 27−33 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 35−38 in Algorithm 3).

If all the edges starting from τ are visited, then the stack top node $\texttt{ST}\langle\texttt{topElement}\rangle$ is popped as successor node of τ to be appended to $\mathbb{T}$ (Line 40−41 in Algorithm 3).

Algorithm 3. getTraceSuccNode(τ, B, **BBInfoList**, ST, $\mathbb{T}$, $\mathbb{BE}$, $\mathbb{NE}$)

Input: Trace tail node τ, CFG entry node B, Basic block information **BBInfoList**, Node stack ST, Candidate trace $\mathbb{T}$, Backedge set $\mathbb{BE}$, Non-backedge set $\mathbb{NE}$
Output: Candidate trace successor node τ (overwritten)

1: $\mathbb{D}_{\mathbb{BE}} \leftarrow \varnothing$, $\mathbb{D}'_{\mathbb{BE}} \leftarrow \varnothing$, $\mathbb{D}_{\mathbb{NE}} \leftarrow \varnothing$, $\mathbb{D}'_{\mathbb{NE}} \leftarrow \varnothing$, $\mathbb{S}_{\mathbb{NE}} \leftarrow \varnothing$
2: **if** $\exists\ \texttt{be} \in \mathbb{BE}, \tau = be.srcNode$ **then**
3: $\quad\mathbb{D}_{\mathbb{BE}} = \{\texttt{be.destNode} \mid \texttt{be} \in \mathbb{BE}, \tau = \texttt{be.srcNode}\}$
4: $\quad$**foreach** $d_{be} \in \mathbb{D}_{\mathbb{BE}}$ **do** ▷ *get candidate nodes that are not visited over*
5: $\quad\quad$**if** $d_{be}.unvisitedCount\langle 0\rangle \%\ d_{be}.bounds\langle 0\rangle \neq 0$ **then**
6: $\quad\quad\quad\mathbb{D}'_{\mathbb{BE}} \leftarrow \mathbb{D}'_{\mathbb{BE}} + \{d_{be}\}$
7: $\quad$**if** $\mathbb{D}'_{\mathbb{BE}} \neq \varnothing$ **then**
8: $\quad\quad$**if** $\tau \in \mathbb{D}'_{\mathbb{BE}}$ **then** ▷ *there is a backedge from τ to itself*
9: $\quad\quad\quad$**return** τ ▷ *τ is not visited over at its current loop level*
10: $\quad\quad$**else**
11: $\quad\quad\quad$**foreach** $d'_{be} \in \mathbb{D}'_{\mathbb{BE}}$ **do**
12: $\quad\quad\quad\quad\mathbb{P}_{\mathbb{B}} \leftarrow$ getNodesInPath(d'_{be}, τ)
13: $\quad\quad\quad\mathbb{I}_{\mathbb{BE}} \leftarrow \mathbb{D}'_{\mathbb{BE}} \cap \mathbb{P}_{\mathbb{B}}$
14: $\quad\quad\quad$**if** $\mathbb{I}_{\mathbb{BE}} \neq \varnothing$ **then**
15: $\quad\quad\quad\quad$**return** $\mathbb{I}_{\mathbb{BE}}\langle 0\rangle$ ▷*return the closest-to-τ node*
16: **else if** $\exists\ \texttt{ne} \in \mathbb{NE}, \tau = ne.srcNode$ **then**
17: $\quad\mathbb{D}_{\mathbb{NE}} = \{\texttt{ne.destNode} \mid \texttt{ne} \in \mathbb{NE}, \tau = \texttt{ne.srcNode}\}$
18: $\quad$**foreach** $d_{ne} \in \mathbb{D}_{\mathbb{NE}}$ **do** ▷ *get the closest-to-τ non-backedge nodes*
19: $\quad\quad\mathbb{P}_{\mathbb{N}} \leftarrow$ getNodesInPath(τ, d_{ne})
20: $\quad\quad$**if** $\mathbb{D}_{\mathbb{NE}} \cap \mathbb{P}_{\mathbb{N}} = \varnothing$ **then**
21: $\quad\quad\quad\mathbb{D}'_{\mathbb{NE}} \leftarrow \mathbb{D}'_{\mathbb{NE}} + \{d_{ne}\}$
22: $\quad$**if** $\mathbb{D}'_{\mathbb{NE}} \neq \varnothing$ **then**
23: $\quad\quad$**foreach** $d'_{ne} \in \mathbb{D}'_{\mathbb{NE}}$ **do** ▷ *get nodes in other backedges*
24: $\quad\quad\quad$**if** $\exists\ be^n \in \mathbb{BE}$, $d'_{ne} = be^n.srcNode$ **then**
25: $\quad\quad\quad\quad\mathbb{S}_{\mathbb{NE}} \leftarrow \mathbb{S}_{\mathbb{NE}} + \{d'_{ne}\}$
26: $\quad\quad s_{ne} = (\mathbb{S}_{\mathbb{NE}} \neq \varnothing)\ ?\ \mathbb{S}_{\mathbb{NE}}\langle 0\rangle\ :\ \mathbb{D}'_{\mathbb{NE}}\langle 0\rangle$ ▷ *candidate successor*
27: $\quad\quad$**if** $\texttt{ST} \neq \varnothing$ **then**
28: $\quad\quad\quad\mathbb{P}_{\mathbb{S}} \leftarrow$ getNodesInPath(B, s_{ne})
29: $\quad\quad\quad$**if** $ST\langle topElement\rangle \in \mathbb{P}_{\mathbb{S}}$ **then**
30: $\quad\quad\quad\quad\tau \leftarrow \texttt{ST}\langle\ \texttt{topElement}\ \rangle, \texttt{ST}.pop()$
31: $\quad\quad\quad$**else** ▷ *the stack top node denotes another branch path*
32: $\quad\quad\quad\quad\tau \leftarrow s_{ne}$ ▷ *but the current path is not visited over*
33: $\quad\quad\quad$**return** τ
34: $\quad\quad$**else** ▷ *the current path is the last path of the current branch*
35: $\quad\quad\quad\mathbb{D}'_{\mathbb{NE}} \leftarrow \mathbb{D}'_{\mathbb{NE}} - \{\texttt{s}_{\texttt{ne}}\}$
36: $\quad\quad\quad$**foreach** $d''_{ne} \in \mathbb{D}'_{\mathbb{NE}}$ **do** ▷ *store remaining header nodes*
37: $\quad\quad\quad\quad\texttt{ST}.push(d''_{ne})$
38: $\quad\quad\quad$**return** s_{ne}
39: **else**
40: $\quad\tau \leftarrow \texttt{ST}\langle\texttt{topElement}\rangle, \texttt{ST}.pop()$
41: $\quad$**return** τ

4.2.3 Cache Behavior Analysis

We use micro-benchmarks to obtain the cache hit and miss latencies of the local, constant, and global memory accesses. When handling the constant and the global memory accesses, the Streaming Multi-processor (SM) on the GPU first tries 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 [45], [46] to obtain the detailed cache configurations, such as the cache size, the cache line size, and the cache associativity.

Given the input workload and NDRange configurations, for each memory access in the execution trace, we obtain the memory referencing address of the kernel arguments from the static analysis. With this information, we analyze the number of memory transactions that a warp would perform for each 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 [47].

5 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 NDRange configurations, we construct an IR instruction pipeline and then simulate the trace on the pipeline in granularity of a round of active work groups.

5.1 IR Instruction Pipeline

5.1.1 Determining the Number of Active Work Groups

Given a kernel with NDRange configuration as global work size GS and work-group size wg. Each work item consumes N_{reg} on-chip registers and N_{sm} bytes shared memory. The number of active work groups N_{awg} per 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

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

$$N_{warp\_per\_wg} = \left\lceil \frac{wg}{T_{warp}} \right\rceil, \tag{2}$$

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

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

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

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

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

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

5.1.2 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 throughput of the target GPU [48]. We consider the basic arithmetic operations 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.

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 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".

5.2 Calculating the Trace Simulation Time

Given the kernel execution trace **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 maneuver 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

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

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

The simulation is implemented with a group of active warps continually consuming the arithmetic and memory access operations in presence of shared resource and cache contention. For preliminary study, we assume the warps are scheduled in a round-robin policy, while other warp scheduling policies, such as greedy then oldest (GTO), can also be adapted easily via modifying the simulated warp scheduling algorithm. Investigating the effect of adopting different warp scheduling policies on the performance estimation is one aspect of the future work. For each memory access, we assume memory 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 which the sum is the profiled cache hit or miss latency.

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 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.

5.3 Discussion and Summary

Table 1 summarizes the parameters used in our performance estimation method. 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 NDRange configurations are fed to the simulator to obtain the estimated results. Moreover, only a round of active work groups is actually used and therefore the simulation time cost is small.

During the simulation, our performance model follows the modular design principle, i.e., each computation and memory resource (such as the different caches, the FPU, and the SFU) is abstracted as one module and then integrated to

TABLE 1
Summary of the Parameters Used for the Performance Estimation

No.	Parameter	Definition	Obtained
1	GS	Global work size	Program configuration
2	wg	Work-group 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)

generate the estimated results. In this way, the performance model can be smoothly applied to new-generation GPUs via adding new features into the simulation part.

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 source-level 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 the future work.

6 Work-Group Size Pruning

The design pruning module targets the effect of selecting different work-group sizes on the runtime performance of the OpenCL kernels. In OpenCL, the input workload is represented as a number of work items and this total number of work items (or the global size, GS) can be divided into several work groups. Each group of work items with a work-group size wg, is a set of work instances mapped to a single SM on GPUs. The design pruning module is intended to solve the problem that can be stated as: Given a kernel with NDRange configuration of global size GS and a set of possible work-group sizes $WG = \{wg_0, wg_1, \ldots, wg_K\}$, where K is the number of possible and valid designs, find the appropriate work-group size wg_{opt} so that the kernel execution time T_{kernel} is minimal.

For Nvidia GPUs, all threads in one warp consume the same instructions, while the consumed instructions in different warps may vary. Due to resource and architectural limit, given a specific work-group size (wg), the number of active warps (N^{wg}_{aw}) for a kernel can be determined at compile time. In general, a larger value of N^{wg}_{aw} delivers better performance, though the best case execution does not always reveal the largest value of N^{wg}_{aw}. Given the number of active work groups N_{awg} for a specific work-group size wg, the number of active warps N^{wg}_{aw} is calculated as

$$N^{wg}_{aw} = N_{awg} \times \left\lceil \frac{wg}{T_{warp}} \right\rceil, \tag{9}$$

where T_{warp} is the number of threads per warp. Therefore Equation (8) can be rewritten as

$$T_{kernel} = \underbrace{T^{spec(\texttt{LAT},\texttt{cacheMissInfo})}_{pipeline(N^{wg}_{aw},\mathbf{T})}}_{\text{Sub-item}\,\textcircled{1}} \times \underbrace{\frac{\lceil \frac{GS}{wg \times N_{SM}} \rceil \times \lceil \frac{wg}{T_{warp}} \rceil}{N^{wg}_{aw}}}_{\text{Sub-item}\,\textcircled{2}}. \tag{10}$$

Consequently, the minimization of the kernel execution time T_{kernel} is converted to the minimization of the sub-items in Equation (10).

6.1 Duplicated Trace Pruning

From Equation (10), we have two observations: ① The latency of the arithmetic and memory access operations (`LAT`), the number of work items per warp (T_{warp}), and the total number of SMs (N_{SM}) are hardware-dependent and

therefore remain constant when varying the work-group size. ② Once the work-group size is given, the execution traces of the kernels are fixed, which indicates that the sequence of IR instructions one warp would consume is known. Note that the GPU occupancy is also fixed since the number of active warps (N_{aw}^{wg}) is determined at compile time. Hence, the cache miss information in the execution trace (`cacheMissInfo`) is determined by the in-memory layout of the input kernel arguments (denoted as `argMemLayout`) and the hardware specifications of the memory hierarchy (denoted as `gpuMemConfig`).

With the above observations, when varying the work-group size wg, the kernel execution time is only determined by `argMemLayout`, N_{aw}^{wg}, and $\mathbf{T}$, since the remaining parameters are hardware-dependent and constant.

We compare the details of `argMemLayout` and $\mathbf{T}$ for each work-group size and combine the work-group sizes with the same contents of `argMemLayout` and $\mathbf{T}$ into one batch. Note that, the same execution traces $\mathbf{T}$ means that both the execution trace for each warp and the number of active warps are identical. We extract `argMemLayout` from the invoked argument list of the kernel function, and obtain the memory size and data type of each argument from the host source code. We observe that for most of the kernels, `argMemLayout` remains constant when varying the work-group size, as it is only affected by the input workload. For the cases in which the memory buffer sizes of kernel arguments are determined by the work-group size, we treat them as work-group sizes in different batches.

The work-group sizes in the same batch show the same kernel execution behavior for a round of active warps, so the values of Sub-item ❶ in Equation (10) are equivalent. Sub-item ❷ in Equation (10) indicates the number of rounds of active warps needed for the entire kernel run. Therefore, the work-group size with the lowest value of Sub-item ❷ in a batch reveals the minimal kernel execution time, and we choose this design as the representative design from this batch. After all the representative designs from all batches are collected, they are fed to the following steps for further design pruning.

6.2 Inferior Pipeline Elimination

The *duplicated trace pruning* step filters out the work-group sizes that showcase the same kernel execution behavior within a round of active warps. In this step, the execution times among different execution traces are analyzed, and designs with inferior pipelines are eliminated.

Algorithm 4 gives the detail of this procedure. For each work-group size wg, we deduce the lower bound of the best-case pipeline execution latency (L_B^{wg}), and thereby calculate the lower bound of the best-case execution time (lat_B^{wg}). If this lower bound is still larger than the upper bound of the worst-case execution time yielded by another design ($lat_W^{wg'}$) within the representative work-group size set, we say that the selected design results in an inferior pipeline and therefore is removed from the search space.

As shown in Line 5 and 8 in Algorithm 4, lat_B^{wg} and $lat_W^{wg'}$ are calculated in a similar way, i.e., by replacing Sub-item ❶ in Equation (10) with L_B^{wg} and $L_W^{wg'}$, respectively. Therefore, in the following we give details of how L_B^{wg} and L_W^{wg} are deduced (Line 4 and 7 in Algorithm 4).

Algorithm 4. Inferior Pipeline Elimination

Input: Representative work-group size set $\mathbb{WG}$, Global size GS, Total number of SMs N_{SM}
Output: Pruned work-group size set **WG**

1: **WG** $\leftarrow \varnothing$
2: **foreach** $wg \in \mathbb{WG}$ **do**
3: `isPruned` $\leftarrow FALSE$
4: $L_B^{wg} \leftarrow$ calcPipelineLowerBoundLatency(wg)
5: $lat_B^{wg} \leftarrow \lceil \frac{GS}{wg \times N_{SM}} \rceil \times \lceil \frac{wg}{T_{warp}} \rceil \times \frac{L_B^{wg}}{N_{aw}^{wg}}$
6: **foreach** $wg' \in \mathbb{WG}$ **do**
7: $L_W^{wg'} \leftarrow$ calcPipelineUpperBoundLatency(wg')
8: $lat_W^{wg'} \leftarrow \lceil \frac{GS}{wg' \times N_{SM}} \rceil \times \lceil \frac{wg'}{T_{warp}} \rceil \times \frac{L_W^{wg'}}{N_{aw}^{wg'}}$
9: **if** $lat_B^{wg} < lat_W^{wg'}$ **then**
10: `isPruned` $\leftarrow TRUE$
11: **break**
12: **if** `isPruned` $== FALSE$ **then**
13: $wg \leftarrow$ **WG**
14: **return WG**

6.2.1 Preliminaries

For Nvidia GPUs, the resources shared by a warp pipeline are the Floating Point Unit (FPU), the Special Function Unit (SFU), and various memory load/store units. Since we model the OpenCL kernel execution at the IR level, the types of memory access are differentiated as memory access from Local Memory (LM), Constant Memory (CM), and Global Memory (GM). Private memory access is performed on-chip and is not mutually exclusive for the work items in one work group. Moreover, the cache accesses of CM and GM load/store are considered as memory accesses from the Constant Cache (CC) and the Global Cache (GC). As mentioned in Section 5.1.2, the memory latency is modeled as three parts: pre-waiting latency, memory delay (denoted as D_{mem}) and post-waiting latency, of which the sum is the micro-benchmarked memory latency (denoted as L_{mem}).

Given an arbitrary execution trace running on a warp pipeline, the mutually exclusive resources are the computation components (FPU, SFU) and the memory components (LM, CM, GM, CC, GC). Suppose the execution trace is defined as $\mathbf{T} = \{\tau_0, \tau_1, \ldots, \tau_m\}$, where τ_i is the execution trace that is the same for N_i warps and $\sum_{i=0}^{m} N_i = N_{aw}^{wg}$. Let's consider the best and worst cases of the kernel execution.

6.2.2 Best-Case Execution Analysis

In the best-case execution, computation and memory access instructions are consumed in an interleaved manner, so that the execution times are overlapped as much as possible. For instance, Fig. 2 presents two extreme cases in which the memory access latency in Case I and the computation latency in Case II overlap perfectly. In these two cases, the overall kernel execution time is equal to either the computation latency or the memory access latency. Therefore, L_B^{wg} is calculated as

$$L_B^{wg} = max(L_{fpu}^{N_{aw}^{wg}}, L_{sfu}^{N_{aw}^{wg}}, Lb_{lm}^{N_{aw}^{wg}}, Lb_{cm}^{N_{aw}^{wg}}, Lb_{gm}^{N_{aw}^{wg}}, Lb_{cc}^{N_{aw}^{wg}}, Lb_{gc}^{N_{aw}^{wg}}), \quad (11)$$

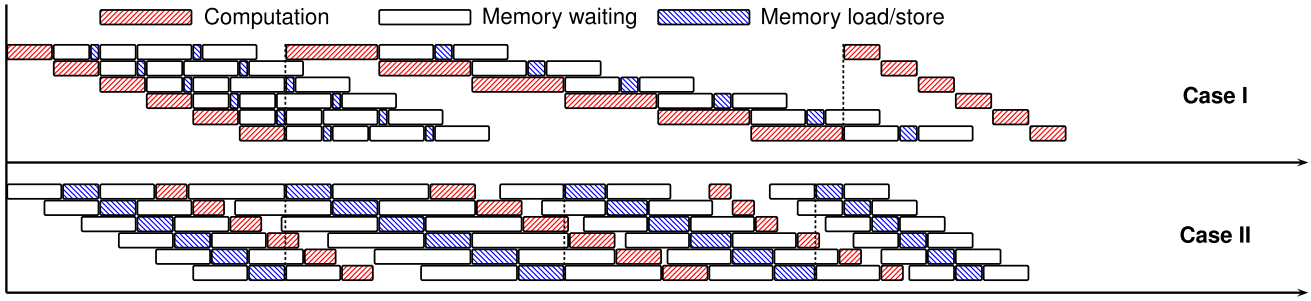


Fig. 2. Best case of the pipeline execution.

where the latencies of the computation components are calculated as follows:

$$L_{comp}^{N_{aw}^{wg}} = \sum_{i=0}^{m} L_{comp}^{N_i} \tag{12}$$

$$L_{comp}^{N_i} = \sum^{opType(\tau_i)} num(OP, \tau_i) \times lat(OP), \tag{13}$$

where $comp = \{fpu, sfu\}$, $opType(\tau_i)$ is the number of instruction types in the execution trace τ_i, $num(OP, \tau_i)$ is the number of OP instructions in the execution trace τ_i, and $lat(OP)$ is the micro-benchmarked latency of OP instruction. Here OP refers to basic arithmetic instructions for the FPU, and transcendental instructions for the SFU, respectively.

For the memory components, the overall latency is directly related to the pipeline depth, i.e., the number of active warps running on the pipeline. This is because memory delay, instead of memory latency, is the main factor that stalls the memory component pipeline. To illustrate this, consider the two cases in Fig. 3, where the pipeline depth d influences the latency of the memory components. In Case I, the pipeline is not deep enough, so the memory load/store delay time cannot hide the memory waiting time ($d \times D_{mem} \leq L_{mem}$). Suppose each warp has κ memory components, the pipeline latency in this case is calculated as

$$L_I = \kappa \times L_{mem} + (d-1) \times D_{mem}. \tag{14}$$

However, as the pipeline becomes deeper, as shown in Case II, the memory delay becomes the dominant factor in the pipeline latency and the memory waiting time is totally overlapped with it ($d \times D_{mem} > L_{mem}$). In this case, the pipeline latency is

$$L_{II} = \kappa \times d \times D_{mem} + L_{mem} - D_{mem}. \tag{15}$$

From Equations (14) and (15), the latency of each type of the memory components is

$$L_{comp}^{N_i} = \kappa^{N_i} \times l_{max}^{N_i} + l_{min}^{N_i} - D_{mem}^{comp} \tag{16}$$

$$l_{max}^{N_i} = max(N_i \times D_{mem}^{comp}, L_{mem}^{comp}) \tag{17}$$

$$l_{min}^{N_i} = min(N_i \times D_{mem}^{comp}, L_{mem}^{comp}), \tag{18}$$

where $comp = \{lm, cm, gm, cc, gc\}$, $i = 0, 1, \ldots, m$, and κ^{N_i} is the number of memory components in the execution trace τ_i. The best-case memory component latency for all the execution traces in a round of active warps is therefore calculated as

$$Lb_{comp}^{N_{aw}^{wg}} = max(L_{comp}^{N_0}, L_{comp}^{N_1}, \ldots, L_{comp}^{N_m}), \tag{19}$$

where $comp = \{lm, cm, gm, cc, gc\}$.

6.2.3 Worst-Case Execution Analysis

In the worst-case execution, the IR instructions in the execution traces construct a permutation that the latencies of the computation and memory components barely overlap. We derive the upper bound of the pipeline latency as an extreme case in which at any time only one IR instruction in the execution trace is consumed as if there is always synchronization between each IR instruction. Consequently, L_W^{wg} is deduced as

$$\begin{aligned} L_W^{wg} = {} & L_{fpu}^{N_{aw}^{wg}} + L_{sfu}^{N_{aw}^{wg}} + Lw_{lm}^{N_{aw}^{wg}} + Lw_{cm}^{N_{aw}^{wg}} \\ & + Lw_{gm}^{N_{aw}^{wg}} + Lw_{cc}^{N_{aw}^{wg}} + Lw_{gc}^{N_{aw}^{wg}} \end{aligned} \tag{20}$$

$$Lw_{comp}^{N_{aw}^{wg}} = \sum_{i=0}^{m} L_{comp}^{N_i}, \tag{21}$$

where $comp = \{lm, cm, gm, cc, gc\}$, $L_{fpu}^{N_{aw}^{wg}}$ and $L_{sfu}^{N_{aw}^{wg}}$ are calculated from Equation (12), and $L_{comp}^{N_i}$ is calculated from Equation (16).

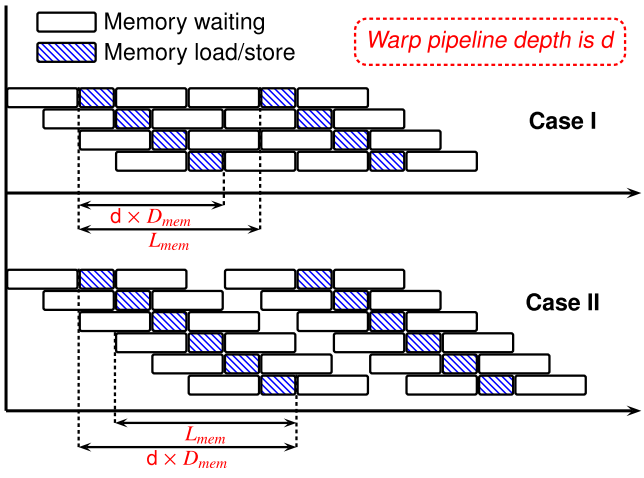


Fig. 3. Sample cases of the memory component pipeline.

6.3 IR Pipeline Simulation

The pruned designs output by the *inferior pipeline elimination* step are a part of the possible work-group sizes in the entire program design space. Subsequently, the trace-based simulation (details illustrated in Section 5) is performed to predict the kernel execution time needed for each design. After the

TABLE 2
Hardware Specifications of the Evaluated GPUs

Device 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	1,058
GeForce 940M	Maxwell GM108	3/384	1,072

exhaustive simulation search, the work-group size that produces the minimum predicted execution time is chosen as the optimal work-group size wg_{opt}.

6.4 Discussion

As can be seen, the design pruning module combines the static execution trace and pipeline analysis results to prune the work-group size configurations and then identifies the optimal or near-optimal work-group size via dynamical execution trace simulations. Therefore, our method does not require any program runs. Unlike comparable state-of-the-art methods, our method neither requires tedious profiling to obtain representative execution results, nor does it adopt performance metrics extracted from the high-level kernel source code.

Although the simulation module performs an exhaustive simulation search of the pruned designs from the static analysis module, this time cost is rather small compared with the exhaustive simulation search of all the possible designs from the entire program design space, because the two-stage static pruning step filters out most of the redundant designs. The bound analysis of the best- and worst-case executions of the IR instruction pipeline manifests the extreme cases of the pipeline latency overlapping, given arbitrary execution traces and arbitrary pipeline depth. Although the actual execution of the pipeline can always be determined once the execution traces are fixed, this information can only be obtained through the dynamical simulations. Hence, the bound estimation of the pipeline latency is intended to effectively reduce the number of total simulated designs at pre-simulation stage.

7 Experiment

7.1 Setup

We used four COTS Nvidia GPUs to evaluate the framework and the detailed information is shown in Table 2. These GPUs are from Kepler and Maxwell architectures with different compute capacities so as to demonstrate the robustness of our framework. We evaluated the proposed framework with a set of OpenCL kernels from the Rodinia [14] benchmark. The simulations were performed on a desktop with an Intel® Core™ i7-3770 CPU.

For the performance estimation, we used the default input from the benchmarks and conducted a design space exploration that results in a total of 306,558 estimation runs. For the design pruning, we used the default input workloads from the benchmarks and excluded the kernels whose work-group size cannot be varied when the global size is fixed. The detailed configuration is given in Table 4. For `backprop`, the number of possible work-group sizes is very small because this application allocates shared memory according to the selected work-group size. Therefore, the shared memory quickly reaches the capacity limit when the work-group size is set to a large value. For `kmeans`, the number of features may vary even if the global size is fixed to (10,000, 1, 1), therefore we tested different cases in which the number of features was set to 16, 32, and 64, respectively.

7.2 Performance Estimation Results

7.2.1 Accuracy

Table 3 presents the accuracy results. The third column in Table 3 lists the number of total designs 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 percent and on each GPU, the optimal kernel prediction can achieve an average MAPE of less than 7 percent. 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 Fig. 4. Due to space limitations, Fig. 4 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 designs 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, our predicted results accurately follow the variation trend of the actual execution time across the design space.

As observed in Fig. 4, the MAPE turns out higher when the actual execution time is a few microseconds, particularly for kernel `nw_kernel1` and `nw_kernel2` (shown in Fig. 4q and 4r). 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. 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 Fig. 4a). For kernel `bpnn_layerforward`, the underestimation in Fig. 4b comes from barrier synchronization and kernel overhead.

TABLE 3
Accuracy and Simulation Time Consumption of the Performance Estimation Results on the Rodinia [14] Benchmark

Bench. name	Kernel name	Number of total designs	Aver. 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	4130.09
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	11751.93
pathfinder	dynproc	31,025	1,469	**3.716**	**6.560**	14.33	9.737	1055.97
Average		15327.9	4148.15	15.39	21.43	14.63	16.71	**931.33**
						17.04		

bfs. The prediction of this application is better than *backprop*, due to the much less branches. As seen in Fig. 4c and 4d, 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 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 (Fig. 4i), which is the same case as kernel `memset` (Fig. 4j). 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 Fig. 4l. The minor underestimation is caused by the kernel overhead, because the execution time of this kernel is less than 60 us.

kmeans. Fig. 4m and 4n 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, Fig. 4q and 4r reveal counter-expectation results. The reason is that the kernel overhead also contributes to the MAPE and it is nonnegligible because the total execution time is only a few microseconds. Consequently, the 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 Fig. 4s, 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 branches are equally visited by the warps.

To summary, our estimation method 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 latency 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 give rather good estimation results.

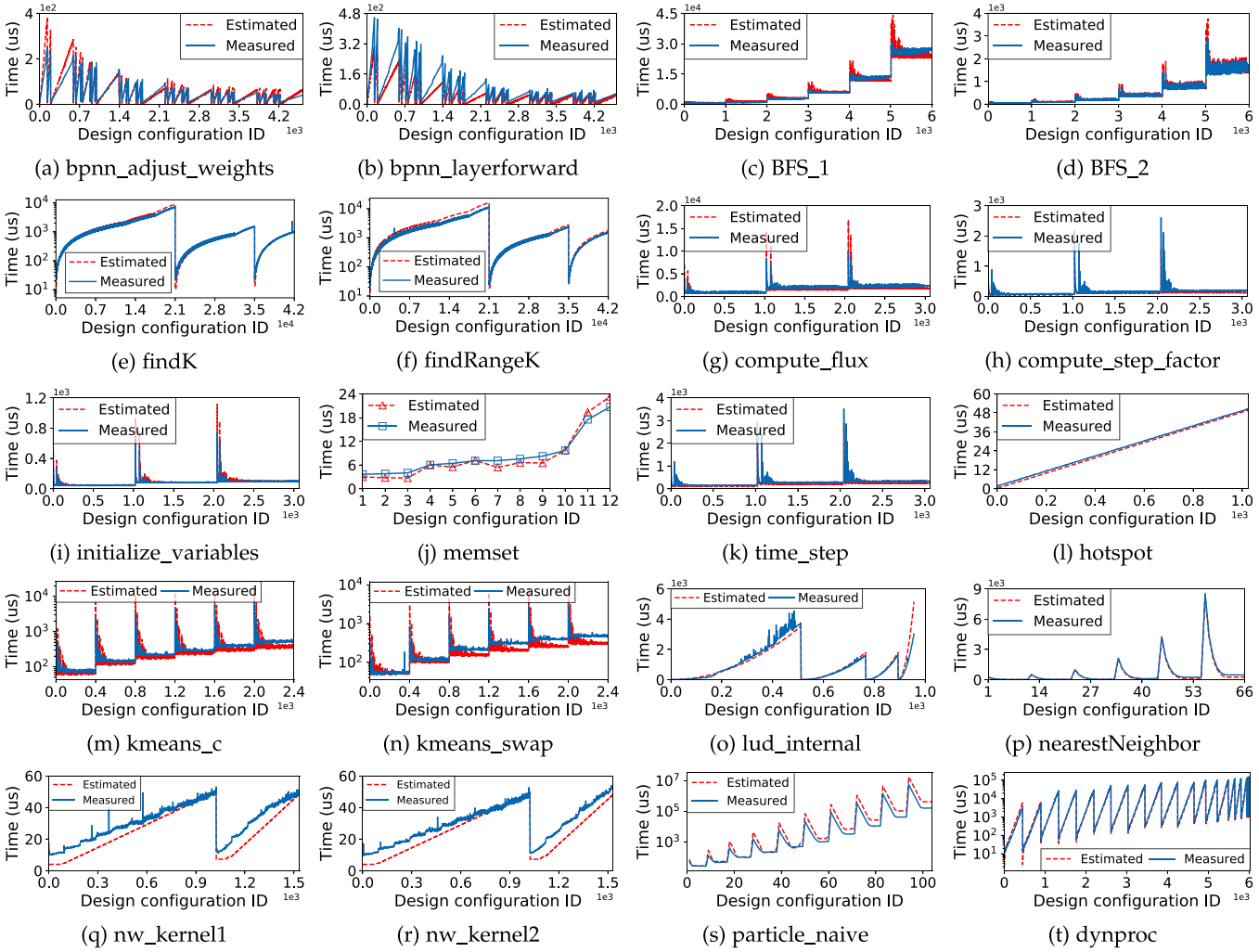


Fig. 4. Comparison of the estimated and measured execution time of the test kernels in Table 3 (Quadro K620).

7.2.2 Simulation Time Cost

The last column in Table 3 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 [49], [50]. 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 [32] and Table 5 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.

7.3 Work-Group Size Pruning Results

7.3.1 Design Space Reduction

Fig. 5 shows the design space reduction results on the test GPUs. The x-axis represents the kernel IDs (Column 2) in Table 4. The bar results show the number of designs for each kernel. Specifically, The `Native` bar represents the number of total designs and the `Pruned` bar represents the number of pruned designs yielded by the two-stage static pruning step. The red points give the design reduction percentage for each test kernel. Note that the number of total designs varies from a few to several thousand and therefore is represented in logarithmic scale.

As can be seen, our approaches are quite effective for most of the test kernels. Specifically, the proposed framework can prune the number of designs by more than 85 percent, for `bfs`, `cfd`, and `nn`. On average, the static pruning step reduces the program design space by 81.87, 78.42, 76.56, and 77.02 percent, respectively, on Quadro K600, GeForce GTX645, Quadro K620, and GeForce 940M GPUs.

For all the GPUs, the limiting factor on the percentage of the program design space reduction is `backprop` and `particleFilter`. As for `backprop`, the number of total designs is rather small (less than 100 for each work-group size configuration and for each GPU, as seen in Table 4), so the number of redundant configurations are greatly outnumbered by the remaining kernels. With regard to `particleFilter`, we observe that with given fixed workload, the kernel execution times with different work-group size configurations are almost equal. This indicates that varying the work-group size has little influence on the kernel runtime performance. In this case, when selecting different work-group sizes, the deduced latencies of the best- and worst-case

TABLE 4
Detailed Configurations of the Test OpenCL Kernels Used for the Work-Group Size Pruning

Benchmark	ID	Kernel name	Global size (NDRange Config.)	Number of total designs			
				Quadro K600	GeForce GTX645	Quadro K620	GeForce 940M
backprop	#1	adjustWeight_1	(100, 1,600, 1)	75	84	75	84
	#2	adjustWeight_2	(100, 3,200, 1)	66	75	66	75
	#3	adjustWeight_3	(100, 6,400, 1)	48	66	48	66
	#4	layerForward_1	(100, 1,600, 1)	75	84	75	84
	#5	layerForward_2	(100, 3,200, 1)	66	75	66	75
	#6	layerForward_3	(100, 6,400, 1)	48	66	48	66
bfs	#7	bfs1_1	(1,048,576, 1, 1)	1,024	1,024	1,024	1,024
	#8	bfs1_2	(2,097,152, 1, 1)	1,024	1,024	1,024	1,024
	#9	bfs2_1	(1,048,576, 1, 1)	1,024	1,024	1,024	1,024
	#10	bfs2_2	(2,097,152, 1, 1)	1,024	1,024	1,024	1,024
cfd	#11	computeFlux_1	(97,152, 1, 1)	1,024	1,024	1,024	1,024
	#12	computeFlux_2	(193,536, 1, 1)	1,024	1,024	1,024	1,024
	#13	computeFlux_3	(232,704, 1, 1)	1,024	1,024	1,024	1,024
	#14	computeStepFactor_1	(97,152, 1, 1)	1,024	1,024	1,024	1,024
	#15	computeStepFactor_2	(193,536, 1, 1)	1,024	1,024	1,024	1,024
	#16	computeStepFactor_3	(232,704, 1, 1)	1,024	1,024	1,024	1,024
	#17	initVariable_1	(97,152, 1, 1)	1,024	1,024	1,024	1,024
	#18	initVariable_2	(193,536, 1, 1)	1,024	1,024	1,024	1,024
	#19	initVariable_3	(232,704, 1, 1)	1,024	1,024	1,024	1,024
	#20	timeStep_1	(97,152, 1, 1)	1,024	1,024	1,024	1,024
	#21	timeStep_2	(193,536, 1, 1)	1,024	1,024	1,024	1,024
	#22	timeStep_3	(232,704, 1, 1)	1,024	1,024	1,024	1,024
gaussian	#23	fan1	(4,096, 1, 1)	1,024	1,024	1,024	1,024
	#24	fan2	(4,096, 4096, 1)	7,262	7,262	7,262	7,262
kmeans	#25	kmeansC_feat16	(10,000, 1, 1)	1,024	1,024	1,024	1,024
	#26	kmeansC_feat32	(10,000, 1, 1)	1,024	1,024	1,024	1,024
	#27	kmeansC_feat64	(10,000, 1, 1)	1,024	1,024	1,024	1,024
	#28	kmeansSwap_feat16	(10,000, 1, 1)	1,024	1,024	1,024	1,024
	#29	kmeansSwap_feat32	(10,000, 1, 1)	1,024	1,024	1,024	1,024
	#30	kmeansSwap_feat64	(10,000, 1, 1)	1,024	1,024	1,024	1,024
nn	#31	nearestNeighbor_1	(131,072, 1, 1)	1,024	1,024	1,024	1,024
	#32	nearestNeighbor_2	(262,144, 1, 1)	1,024	1,024	1,024	1,024
	#33	nearestNeighbor_3	(524,288, 1, 1)	1,024	1,024	1,024	1,024
	#34	nearestNeighbor_4	(1,048,576, 1, 1)	1,024	1,024	1,024	1,024
particleFilter	#35	particleNaive	(2,048, 1, 1)	1,024	1,024	1,024	1,024

pipeline executions are comparable to each other. Therefore, the static analysis module is not able to significantly prune the design configurations, resulting in a still very large pruned design space.

7.3.2 Search Quality

Fig. 6 presents the performance results on the test GPUs. The x-axis represents the kernel IDs (Column 2) in Table 4, and the y-axis represents the ratio of the runtime performance of the test kernels with the selected work-group size to that with the truly optimal design. On average, the performance with the selected designs is 1.132 times slower than that with the truly optimal configurations, for the four GPUs. For the kernel `adjustWeight`, our hybrid search method manages to find the truly optimal work-group size configuration for the Quadro K600, Quadro K620, and GeForce 940 GPUs.

The worst results come from the kernel `layerForward`, `fan2`, and `kmeansSwap`. The reason is that the runtime-dependent branch analysis and memory access in these cases makes the performance model generate pessimistic estimations. Hence, the dynamical simulation fails to accurately predict some extreme designs from which the minimal estimated execution time is deduced. In this case, the assumed conservative design is chosen as the estimated optimal work-group size and the performance of this estimated optimal configuration is worse than that of the truly optimal design. To demonstrate the effectiveness of the static analysis module, for the aforementioned kernels, we exhaustively executed the test kernels with pruned designs and chose the work-group size with the minimal execution time as a candidate work-group size configuration. Results show that the candidate configurations always match the truly optimal work-group size.

To showcase the necessity of the work-group size pruning, we present the percentile of the estimated best design in the entire design space in terms of the kernel execution time. The results are shown in Fig. 7. On average, the kernel performance with the estimated optimal work-group size outperforms 87.15, 85.35, 78.84, and 62.98 percent of all the design configurations, for Quadro K600, GeForce GTX645, Quadro K620, and GeForce 940M GPUs,

TABLE 5
Comparison of the Simulation Time Costs of GPGPU-Sim [32] and our Framework

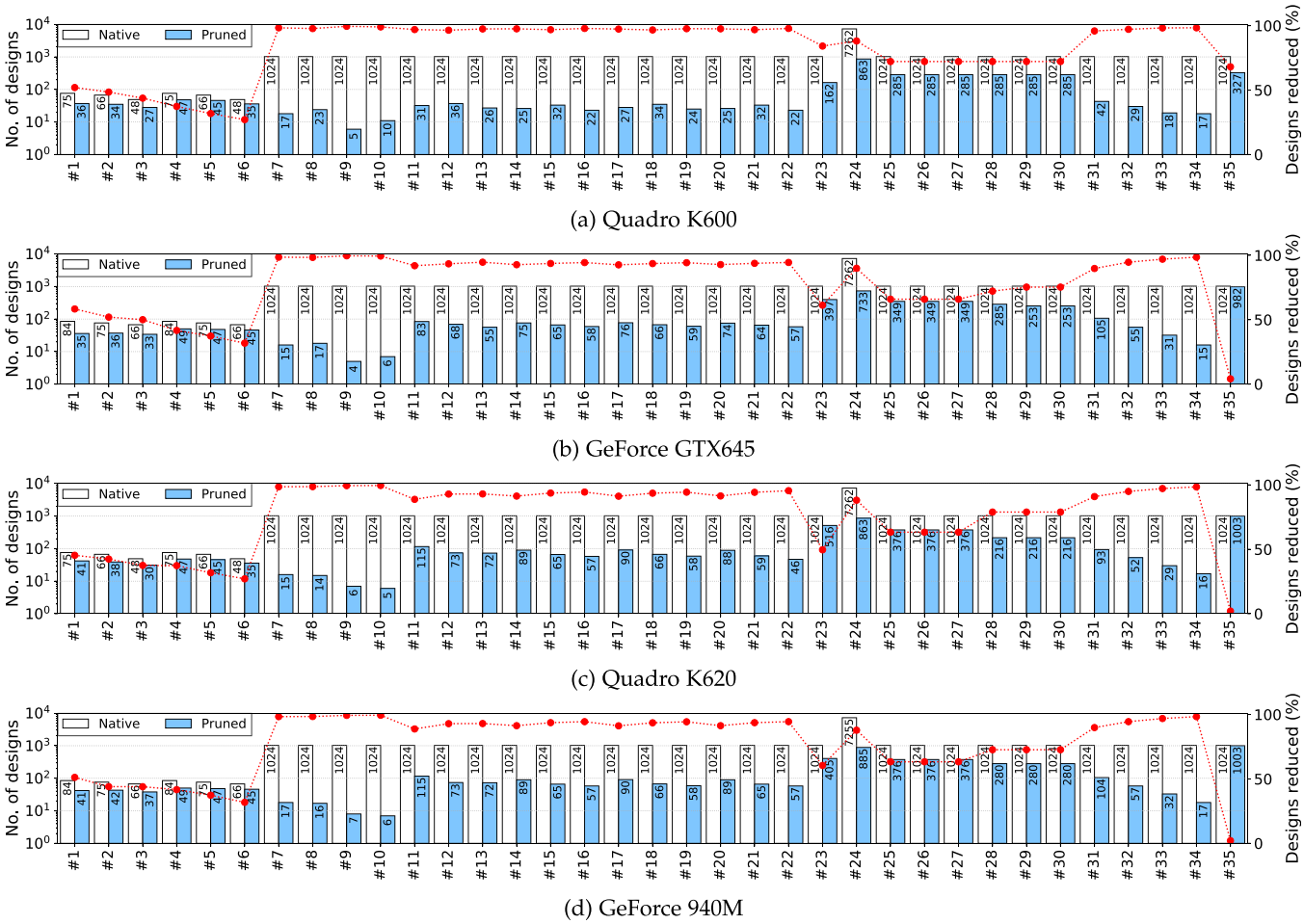
Benchmark	Simulation time (ms)		Speedup
	GPGPU-Sim	Our framework	
bfs	4,517,000	70.49	64080.01
hotspot	200,000	4130.09	48.43
lud	168,000	45.10	3725.06
nn	3,000	6.030	497.51
nw	1,673,000	129.26	12942.91
pathfinder	280,000	1055.97	265.16
Geo. mean	244433.52	148.69	**164.39**

respectively. The performance of GeForce 940M shows slightly worse result than the other GPUs because of the kernel `initVariable`, `kmeansC_feat64`, `kmeansSwap`, and `particleNaive`. The reason is that for `initVariable`, the kernel execution time turns out a very stable trend across all available designs. In this case, the runtime performance of the estimated optimal design stays in the lower end of the entire program space but the performance slowdown against the truly optimal design is still not very high (this can be observed in Fig. 6). For the rest kernels, the estimated optimal design in the hybrid and the exhaustive search is the same work-group size configuration and the kernel execution time when selecting this design is much larger than that of the truly optimal design (this can be observed in Fig. 6). In this case, the percentile of the estimated best design in the entire design space becomes much lower.

7.3.3 Search Time Cost

We recorded the time needed to obtain the optimal design for the test kernels when using *exhaustive simulation search* and *hybrid search*, respectively. In the exhaustive simulation search scenario, we use the performance estimation method to obtain the execution time of all the available designs and then choose the design with minimal predicted execution time as the selected optimal design. While in the hybrid search scenario, only the designs generated from the work-group size pruning module are fed to the estimation framework to obtain the final estimated optimal design. The results are shown in Fig. 8.

On average, the hybrid search framework takes only 21.84, 25.82, 25.55, and 25.48 percent of the time required by the exhaustive simulation search to find the optimal work-group size, for Quadro K600, GeForce GTX645, Quadro K620, and GeForce 940M GPUs, respectively. Similar to the percentage of the program design space reduction, the limiting factor is the large search time needed for kernel `backprop`, and `particleFilter`.



(a) Quadro K600

(b) GeForce GTX645

(c) Quadro K620

(d) GeForce 940M

Fig. 5. Design space reduction results on the test GPUs.

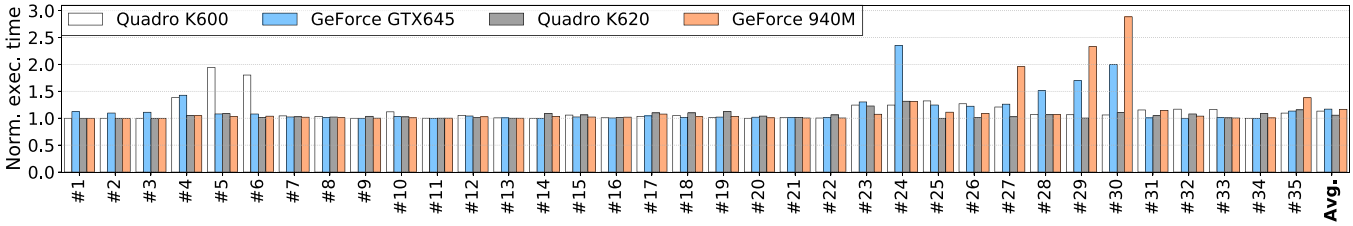


Fig. 6. Normalized execution time of the selected design on the test GPUs.

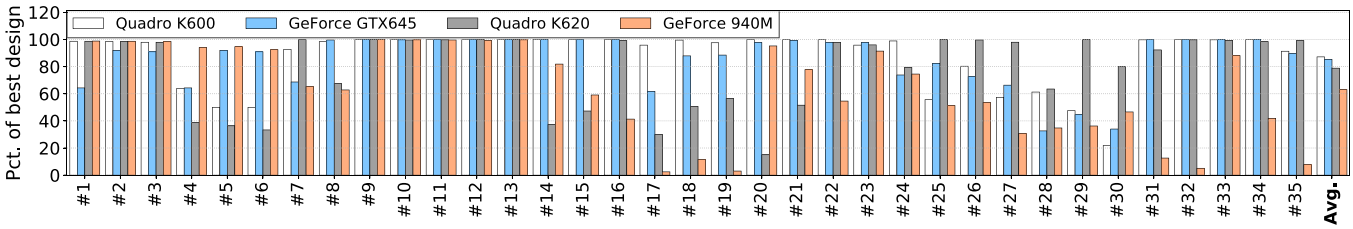


Fig. 7. Percentile of the estimated best design in the total program design space in terms of the kernel execution time.

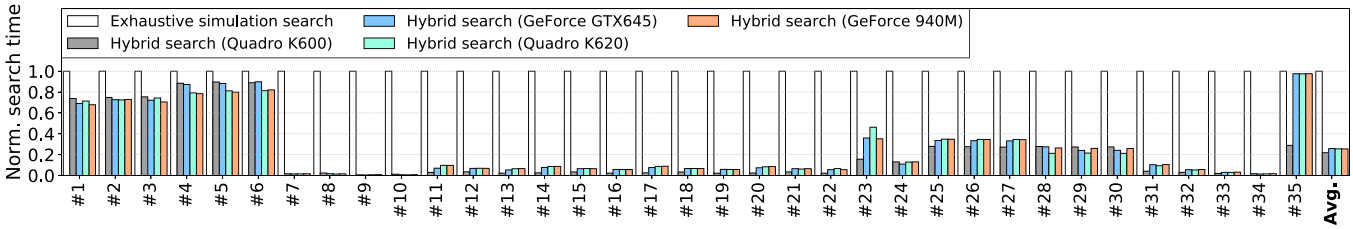


Fig. 8. Normalized time costs of the exhaustive simulation search and the hybrid search on the test GPUs.

8 Conclusion

In this work, we propose a hybrid framework for the performance estimation and work-group size pruning of parallel OpenCL 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. The available work-group sizes of a given fixed workload can be further pruned based on the execution trace and pipeline analysis results, and the hybrid performance estimation method. Our framework requires no prior knowledge about hardware performance counter metrics or pre-executed measurement results. Moreover, it needs no program runs to deduce the execution time and find the optimal or near-optimal work-group size configurations. 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. Furthermore, it can significantly reduce the program design space and generate the estimated work-group size which can deliver runtime performance comparable to that with the truly optimal configuration.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments that greatly contributed to improving the final version of this article. The authors would also like to thank the editors for their generous comments and support during the review process. This work is supported in part by 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 Grants Number NSF-CCF-1657333, NSF-CCF-1717754, NSF-CNS-1717984, and NSF-CCF-1750656.

References

[1] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 725–737.

[2] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 564–576.

[3] K. O'neal, P. Brisk, A. Abousamra, Z. Waters, and E. Shriver, "GPU performance estimation using software rasterization and machine learning," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, 2017, Art. no. 148.

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

[5] Z. Yu *et al.*, "Accelerating GPGPU architecture simulation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 331–332, 2013.

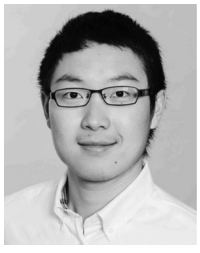
[6] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, "TBPoint: Reducing simulation time for large-scale GPGPU kernels," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 437–446.

[7] C. Nugteren and V. Codreanu, "CLTune: A generic auto-tuner for OpenCL kernels," in *Proc. IEEE 9th Int. Symp. Embedded Multicore/Many-core Syst.-on-Chip*, 2015, pp. 195–202.

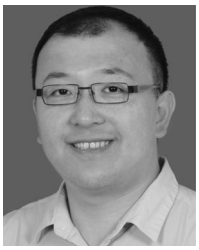
[8] J. Ansel *et al.*, "OpenTuner: An extensible framework for program autotuning," in *Proc. 23rd Int. Conf. Parallel Architectures Compilation*, 2014, pp. 303–316.

[9] T. L. Falch and A. C. Elster, "Machine learning based auto-tuning for enhanced OpenCL performance portability," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, 2015, pp. 1231–1240.

[10] S. Ryoo *et al.*, "Program optimization space pruning for a multithreaded GPU," in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2008, pp. 195–204.
[11] T. Dao and J. Lee, "An auto-tuner for OpenCL work-group size on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 2, pp. 283–296, Feb. 2018.
[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2nd IEEE/ACM Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.
[13] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate GPU performance estimation through source-level analysis and trace-based simulation," in *Proc. IEEE 25th Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 506–518.
[14] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[15] J. Lai and A. Seznec, "Break down GPU execution time with an analytical method," in *Proc. Workshop Rapid Simul. Perform. Eval., Methods Tools*, 2012, pp. 33–39.
[16] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *ACM SIGARCH Comput. Architecture News*, vol. 37, no. 3, pp. 152–163, 2009.
[17] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M. W. Hwu, "An adaptive performance modeling tool for GPU architectures," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 105–114, 2010.
[18] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A performance prediction model for the CUDA GPGPU platform," in *Proc. IEEE 16th Int. Conf. High Perform. Comput.*, 2009, pp. 463–472.
[19] M. Amarís, D. Cordeiro, A. Goldman, and R. Y. de Camargo, "A simple BSP-based model to predict execution time in GPU applications," in *Proc. IEEE 22nd Int. Conf. High Perform. Comput.*, 2015, pp. 285–294.
[20] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[21] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proc. 10th Annu. ACM Symp. Theory Comput.*, 1978, pp. 114–118.
[22] P. B. Gibbons, Y. Matias, and V. Ramachandran, "The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms," *SIAM J. Comput.*, vol. 28, pp. 733–769, 1999.
[23] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 382–393.
[24] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 673–686.
[25] Q. Wang and X. Chu, "GPGPU performance estimation with core and memory frequency scaling," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst.*, 2018, pp. 417–424.
[26] K. Zhou, G. Tan, X. Zhang, C. Wang, and N. Sun, "A performance analysis framework for exploiting GPU microarchitectural capability," in *Proc. Int. Conf. Supercomputing*, 2017, Art. no. 15.
[27] I. Baldini, S. J. Fink, and E. Altman, "Predicting GPU performance from CPU runs using machine learning," in *Proc. IEEE 26th Int. Symp. Comput. Archit. High Perform. Comput.*, 2014, pp. 254–261.
[28] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: A statistical approach," in *Proc. IEEE 6th Int. Conf. Netw. Archit. Storage*, 2011, pp. 149–158.
[29] 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 *Proc. IEEE 15th Int. Symp. Netw. Comput. Appl.*, 2016, pp. 326–333.
[30] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for GPUs with caches," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 7, pp. 1800–1813, Jul. 2015.
[31] C. Gerum, O. Bringmann, and W. Rosenstiel, "Source level performance simulation of GPU cores," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2015, pp. 217–222.
[32] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 163–174.
[33] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for GPGPU," in *Proc. 18th Annu. IEEE/ACM Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2010, pp. 351–360.
[34] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proc. 19th Int. Conf. Parallel Architectures Compilation Techn.*, 2010, pp. 353–364.
[35] R. Balasubramanian *et al.*, "Enabling GPGPU low-level hardware explorations with miaow: An open-source RTL implementation of a GPGPU," *ACM Trans. Architecture Code Optim.*, vol. 12, no. 2, 2015, Art. no. 21.
[36] C. Jiang and M. Snir, "Automatic tuning matrix multiplication performance on graphics hardware," in *Proc. 14th Int. Conf. Parallel Architectures Compilation Techn.*, 2005, pp. 185–194.
[37] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. ACM/IEEE Conf. Supercomputing*, 2007, pp. 1–12.
[38] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. ACM/IEEE Conf. Supercomputing*, 2008, pp. 1–12.
[39] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, Art. no. 30.
[40] R. Lim, B. Norris, and A. Malony, "Autotuning GPU kernels via static and predictive analysis," in *Proc. 46th Int. Conf. Parallel Process.*, 2017, pp. 523–532.
[41] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, 2013, pp. 157–166.
[42] A. B. Hayes, L. Li, D. Chavarría-Miranda, S. L. Song, and E. Z. Zhang, "Orion: A framework for GPU occupancy tuning," in *Proc. 17th Int. Middleware Conf.*, 2016, Art. no. 18.
[43] M. Sinn and F. Zuleger, "LOOPUS: A tool for computing loop bounds for C programs," in *Proc. Workshop Invariant Gener.*, 2010, pp. 185–186.
[44] R. A. Van Engelen, "Efficient symbolic analysis for optimizing compilers," in *Proc. Int. Conf. Compiler Construction*, 2001, pp. 118–132.
[45] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 235–246.
[46] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 72–86, Jan. 2017.
[47] S. Wang, G. Zhong, and T. Mitra, "CGPredict: Embedded GPU performance estimation from single-threaded applications," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, 2017, Art. no. 146.
[48] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic OpenCL device characterization: Guiding optimized kernel design," in *Proc. 17th Int. Eur. Conf. Parallel Process.*, 2011, pp. 438–452.
[49] S. Lee and W. W. Ro, "Parallel GPU architecture simulation framework exploiting work allocation unit parallelism," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2013, pp. 107–117.
[50] G. Malhotra, S. Goel, and S. R. Sarangi, "GpuTejas: A parallel simulator for GPU architectures," in *Proc. IEEE 21st Int. Conf. High Perform. Comput.*, 2014, pp. 1–10.



Xiebing Wang received the BEng and MEng degrees from Computer School, Wuhan University, China, in 2012 and 2015, respectively. Currently he is working towards the PhD degree in the chair of robotics, artificial intelligence and real-time systems, with the Department of Informatics, Technical University of Munich. His research interests include autonomous driving, parallel computing, and heterogeneous computing.



Xuehai Qian received the BSc degree from Beihang University, China, in 2004, the MSc degree from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2007, and the PhD degree from the Computer Science Department, University of Illinois at Urbana-Champaign, in 2013. Currently, he is an assistant professor with the Ming Hsieh Department of Electrical Engineering and the Department of Computer Science, University of Southern California. His research interests include the fields of computer architecture, architectural support for programming productivity, and correctness of parallel programs.

Alois Knoll received the MSc degree from the University of Stuttgart, Germany, in 1985 and the PhD degree from the Technical University of Berlin, Germany, in 1988. He served as the faculty of the Computer Science Department of TU Berlin until 1993. He was a full professor with the University of Bielefeld until 2001. Since 2001 he has been a professor of computer science at the Department of Informatics of Technical University of Munich. Between April 2004 and March 2006 he was executive director of the Institute of Computer Science at TUM. Between 2007 and 2009, he was a member of the EUs highest advisory board for information technology, ISTAG, the Information Society Technology Advisory Group, and a member of its subgroup for Future and Emerging Technologies (FET). His research interests include cognitive, medical and sensor-based robotics, multi-agent systems, data fusion, adaptive systems, multimedia information retrieval, model-driven development of embedded systems with applications to automotive software and electric transportation, as well as simulation systems for robotics and traffic.

Kai Huang received the BSc degree from Fudan University, China, in 1999, MSc degree from the University of Leiden, Netherlands, in 2005, and PhD degree in ETH Zurich, Switzerland, in 2010. He joined Sun Yat-Sen University as a professor, in 2015. He was appointed as the director of the Institute of Unmanned Systems of School of Data and Computer Science, in 2016. He was a senior researcher with the Computer Science Department, Technical University of Munich, Germany from 2012 to 2015 and a research group leader at fortiss GmbH in Munich, Germany, in 2011. He was awarded the Program of Chinese Global Youth Experts 2014 and was granted the Chinese Government Award for Outstanding Self-Financed Students Abroad 2010. He has served as a member of the technical committee on Cybernetics for Cyber-Physical Systems of IEEE SMC Society since 2015. His research interests include techniques for the analysis, design, and optimisation of embedded systems, particularly in the automotive and robotic domains.