Dopia: Online Parallelism Management for Integrated CPU/GPU Architectures

Younghyun Cho
University of California, Berkeley
younghyun@berkeley.edu

Jiyeon Park
Seoul National University
jiyeon@csap.snu.ac.kr

Florian Negele
ETH Zürich
negelef@openbrace.org

Changyeon Jo
Seoul National University
changyeon@csap.snu.ac.kr

Thomas R. Gross
ETH Zürich
trg@inf.ethz.ch

Bernhard Egger
Seoul National University
bernhard@csap.snu.ac.kr

Abstract

Recent desktop and mobile processors often integrate CPU and GPU onto the same die. The limited memory bandwidth of these integrated architectures can negatively affect the performance of data-parallel workloads when all computational resources are active. The combination of active CPU and GPU cores achieving the maximum performance depends on a workload's characteristics, making manual tuning a time-consuming task. Dopia is a fully automated framework that improves the performance of data-parallel workloads by adjusting the **D**egree **O**f **P**arallelism on **I**ntegrated **A**rchitectures. Dopia transparently analyzes and rewrites OpenCL kernels before executing them with the number of CPU and GPU cores expected to yield the best performance. Evaluated on AMD and Intel integrated processors, Dopia achieves 84% of the maximum performance attainable by an oracle.

CCS Concepts • **Computer systems organization** → **Parallel architectures**; • **Software and its engineering** → **Source code generation**; **Runtime environments**; **Concurrency control**.

Keywords Parallelism management, integrated architectures, code analysis, machine learning

ACM Reference Format:
Younghyun Cho, Jiyeon Park, Florian Negele, Changyeon Jo, Thomas R. Gross, and Bernhard Egger. 2022. Dopia: Online Parallelism Management for Integrated CPU/GPU Architectures. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22), April 2–6, 2022, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503221.3508421

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508421

1 Introduction

Many recent desktop and mobile processors integrate CPU and GPU cores onto a single die. Examples of such integrated architectures are AMD's accelerated processing units (APU) [3] and Intel processors since Skylake [11]. This design presents an attractive alternative to dedicated GPUs because of the lower cost and energy consumption [23, 25] and because the shared off-chip memory supports simpler parallel programming models that do not require explicit data copying operations [14, 32].

However, achieving maximal performance for data-parallel workloads on integrated architectures is a surprisingly challenging problem. The characteristics of a parallel workload influences its affinity: workloads with frequent control flow changes and irregular memory accesses tend to achieve better performance on the CPU, while kernels with few control divergences and regular memory access patterns typically achieve higher performance on the GPU [24, 36]. Orchestrating the co-execution of a workload on CPU and GPU cores without a prior profile run is a non-trivial problem [6, 37] since (1) the optimal static partitioning of the workload to the CPU and the GPU is unknown, and (2) a dynamic partitioning scheme must implement a low-overhead global shared queue to distribute the workload dynamically.

Existing techniques select the execution mode based on the kernel's affinity [10, 37, 38]. The state-of-the-art technique [37], e.g., executes a kernel exclusively on the CPU, the GPU, or in parallel on all cores of the CPU and the GPU. Contrary to intuition, however, engaging all available computational cores does not necessarily lead to maximum performance due to the limited shared memory bandwidth. Figure 1 demonstrates this fact with a heatmap showing the throughput of the `Gesummv` benchmark [15] for an input problem size of 16,384 on AMD Kaveri [2]. Normalized to the best configuration that utilizes four CPU threads and 192 GPU threads, the configurations CPU only (CPU: 4, GPU: 0 threads), GPU only (CPU: 0, GPU: 512 threads), and CPU+GPU (CPU: 4, GPU: 512 threads) only achieve 78, 13, and 61 percent of the maximally attainable performance. Even though the `Gesummv` kernel is CPU-affine, additional processing on the GPU may lead to better performance, but only as long as the memory

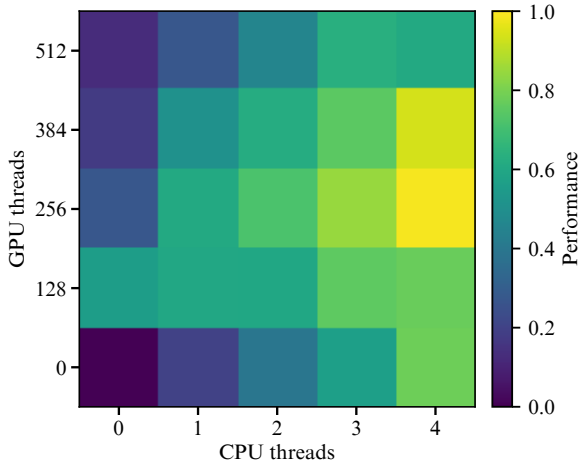


Figure 1. Normalized throughput of `Gesummv` [15] for varying workload partitionings on AMD Kaveri [2]. The X and Y axes show the number of active CPU and GPU threads.

system is not overloaded. If too many GPU threads are active, the outnumbered CPU cores experience a significant performance degradation caused by congestion in the memory system.

Predicting and managing thread-level parallelism and thread placement on multi-core processors has long been an important research topic [5, 8, 12, 13, 29]. Existing techniques, however, are not applicable to integrated architectures because of the integrated architectures' heterogeneity and the black-boxed GPU thread scheduler that does not allow for a dynamic adjustment of the number of active threads.

Our goal is to execute data-parallel workloads at maximum performance on integrated architectures. Unlike in HPC environments where manual software optimization based on profiling data is standard, our method does not require prior profile runs or manual intervention. To achieve this goal, the following research and engineering problems must be addressed: (1) model performance of data-parallel kernels in dependence of the number of active CPU and GPU cores on a given integrated architecture, (2) extract the relevant characteristics from a kernel's source code for the performance model, (3) control the degree of parallelism on the GPU with a pure software approach, and (4) manage a kernel's execution by dynamically distributing a workload to CPU and GPU core resources.

To this end, we present Dopia, a fully automated, software-only technique that automatically adjusts the **D**egree **O**f **P**arallelism of data-parallel workloads on **I**ntegrated **A**rchitectures. Dopia is integrated into the OpenCL runtimes of integrated architectures. A kernel is first submitted to the OpenCL runtime for compilation. At that time, Dopia extracts the kernel's performance-relevant features through static code analysis and generates a CPU and a GPU version that can dynamically adjust their degree of parallelism (DoP) at runtime. When the kernel is launched, Dopia first invokes the machine learning model to predict the number of CPU and GPU cores expected to minimize the kernel's runtime, then executes the kernels on the CPU and the GPU with the selected DoP. During execution of the workload, Dopia's dynamic workload distribution technique dynamically assigns work to CPU and GPU resources to achieve load-balancing.

Dopia is evaluated on an AMD Kaveri and an Intel Skylake integrated architecture. We present a parameterizable synthetic workload that is used to characterize the platforms and generate the machine learning model. We analyze the performance of the dynamic workload distribution and discuss the choice and performance of the selected machine learning technique. An evaluation with fourteen data-intensive real-world OpenCL kernels from Polybench [15], sparse-matrix vector multiplication (SpMV), and a PageRank algorithm shows that Dopia achieves 84% of the maximum performance attainable by an oracle that always predicts the best degree of parallelism for any given kernel. This result includes all runtime overhead incurred by evaluating the model and dynamic workload partitioning of Dopia.

The remainder of this paper is organized as follows. Sections 2 and 3 discuss related work and the background of this work. Section 4 introduces the high-level design of Dopia. Sections 5–7 cover the code analysis, the malleable kernel generation, the machine learning (ML) model, and the workload distribution technique employed by Dopia. Sections 8 and 9 describe the experimental setup and discuss the results, and Section 10 concludes this paper.

2 Related Work

Dopia combines several techniques from software-based parallelism management for GPUs, performance prediction based on machine learning, to dynamic workload partitioning on a heterogeneous architecture. While each topic has been researched independently, to the best of our knowledge, this is the first work to present an integrated and automatic framework for integrated architectures. This section gives an overview of related work.

2.1 Integrated Architectures

Partitioning a workload is key to leveraging different processors on heterogeneous systems. The problem of optimally partitioning a workload and reducing unnecessary data transfers on disjoint memory systems has been investigated in previous projects [22, 24, 31]. The global shared memory on integrated architectures allows for a more efficient data partitioning since no costly data copy operations are required. Kaleem et al. [18] dynamically partition a workload between the CPU and the GPU to achieve load balancing while considering the heterogeneity of the different cores. Zhang et al. [36] and Cho et al. [6] optimize irregular data-parallel

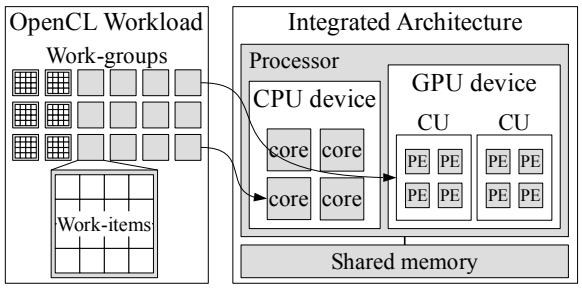


Figure 2. Running OpenCL workloads on integrated architectures.

workloads on integrated CPU-GPU architectures by executing irregular computational work-chunks on the CPU.

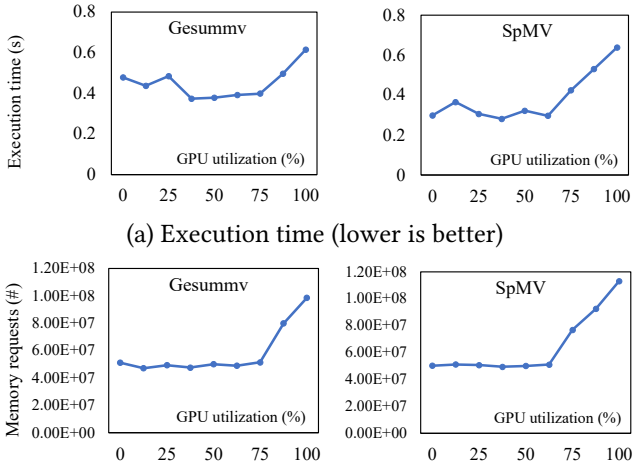
However, several studies [37, 38] confirm that utilizing all available CPU and GPU resources can lead to severe performance degradation on integrated architectures due to the shared memory's limited bandwidth. Researchers have recently focused on selecting the best execution mode between CPU, GPU, and collaborative execution [10, 37]; however, these approaches still leave room for improvement because varying the number of CPU and GPU threads is not considered. The technique presented in this work considers the memory bottleneck of integrated architectures and transparently adjusts the thread-level parallelism on CPU and GPU to achieve maximal performance.

2.2 Parallelism Management

Managing the degree of parallelism has long been an important optimization technique for multi-core processors [5, 9, 12, 29, 34]. Unlike CPU architectures, however, modern GPU hardware architectures do not allow for a flexible adjustment of the thread-level parallelism since the GPU threads are managed directly by a GPU-internal scheduler. By default, GPU schedulers create as many threads as possible to utilize all computational hardware resources of the GPU. Various techniques optimize the thread-level parallelism on GPUs by modifying the GPU scheduler [17, 19], but these techniques require modifications to the hardware and are not readily applicable to existing GPUs. In contrast to existing work, we present a software-based solution to parallelism management for GPU cores.

2.3 Performance Model

Analytical models of application performance as a function of the number of active threads [7, 13, 16, 34, 35] commonly require one or several profiling runs of the workload to extract the necessary performance features. Emani et al. [12] determine the number of threads on multi-core processors based on an ML model; however, the techniques do not consider the degree of parallelism for GPUs or heterogeneous architectures. Dopia does not rely on profile runs. Instead it extracts performance-relevant features of an OpenCL kernel at compile-time through static code analysis. At runtime, Dopia employs a machine learning model to determine the expected best degree of parallelism of CPU and GPU cores.



(a) Execution time (lower is better)

(b) Number of total memory requests

Figure 3. Performance for varying GPU core utilizations and four CPU threads. The number of active PEs across all CUs on AMD Kaveri [2] is adjusted using the code transformation technique presented in Section 6. CPU/GPU workload partitioning is performed by Dopia's dynamic workload distribution algorithm (Section 7). The work-group size is 256.

3 Background and Motivation

3.1 Execution Model

Programming models such as OpenCL [20] or CUDA [26] enable data-level parallel processing on the GPU. Figure 2 illustrates the OpenCL abstraction of a workload and the mapping to an integrated architecture. The *n*-dimensional data-parallel input is split into *work-items*, the smallest atomic unit of work. Work-items are grouped into *work-groups* that constitute the minimal unit of assigned work. On the hardware side, the CPU and the GPU represent independent *compute devices*. A compute device comprises one or more *compute units* (CUs). On the GPU, each CU contains a number of *processing elements* (PEs), also called *GPU cores* in this paper. On the CPU, CUs are a logical concept; typically, one core represents one CU [33]. Thanks to the shared global off-chip memory, work-groups can be assigned to either CPU or GPU CUs without the need for data copies.

3.2 Degree of Parallelism and Performance

The execution time of data-intensive workloads on integrated architectures varies significantly depending on the number of active cores. Figure 3 plots the execution time and the number of memory requests of the Gesummv and the SpMV kernel for an increasing number of active GPU cores on AMD Kaveri [2]. From Figure 3 (a), we observe that the best configuration (37.5% GPU utilization for both Gesummv and SpMV) achieves a significant speedup compared to using all 512 GPU cores. Figure 3 (b) reveals the reason for the slowdown: the increasing number of active GPU cores causes many more capacity misses in the GPU's shared L2

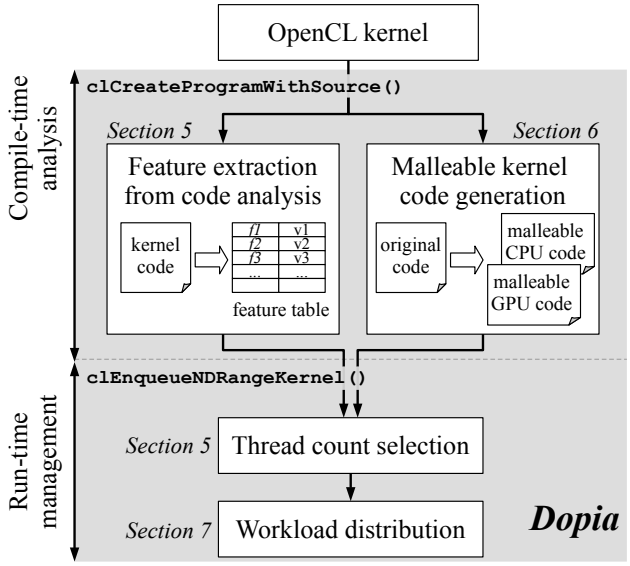


Figure 4. Overview of the Dopia approach.

cache [28], resulting in a significantly higher number of DRAM accesses and congestion in the memory system.

3.3 Controlling the Degree of Parallelism

On the CPU, the number of active cores can easily be controlled by a work-group scheduler that assigns work-groups only to a limited number of threads. On the GPU, the scheduling algorithm is implemented directly on-chip and not controllable from the outside. A global scheduler assigns work-groups to CUs, and the CUs' warp schedulers execute the work-items on all PEs of a CU. Existing work has adjusted the thread-level parallelism on GPUs by modifying the hardware GPU scheduler or not activating all CUs of the GPU [17, 19]. Dopia presents a novel software-only approach that executes a workload on all CUs of the GPU but throttles the number of active PEs within a CU. The advantage of Dopia's technique is that it enables fine-grained control over the GPU's level of parallelism in software.

The main idea of Dopia's thread-level parallelism management on the GPU is to block the execution of work-items on certain PEs *inside a CU*. OpenCL provides no means to identify the index of a PE a kernel is running on; however, a kernel can query the index of a work-item within its work-group using the `get_local_id(0)` function. Since work-items are mapped linearly to the available PEs, and the number of PEs in a CU is known, we can compute the index of a PE by the index of its work-item. It is then possible to use control flow divergence to prevent a PE from executing its assigned work-item. Since the GPU scheduler assumes that all assigned work-items are processed, the active PEs within a CU use a CU-local variable to track the processed work-items and loop until the entire work-group has been processed. The details of our approach are discussed in more detail in Section 6.

4 Overview

Dopia is an additive runtime library running on top of a fully-functional OpenCL runtime system. Through library interpositioning, Dopia transparently intercepts OpenCL API calls and analyzes and transforms OpenCL kernels into malleable code. When executing a workload, Dopia predicts the optimal level of parallelism and orchestrates the workload distribution to the CPU and GPU cores. Figure 4 shows the overall architecture and work-flow of Dopia.

When an OpenCL kernel is compiled, Dopia performs static code analysis and transforms the kernel into malleable code that allows runtime adjustments to its degree of parallelism. The code feature analysis extracts information about ALU and memory operations (number, access patterns). The code transformation rewrites the original OpenCL kernel into a CPU version and a malleable GPU kernel. When the kernel is launched via `clEnqueueNDRangeKernel`, Dopia feeds the extracted code features into a performance model to predict the optimal degree of parallelism of the kernel on the given architecture. The kernel is then executed on the selected number of CPU and GPU cores, and Dopia dynamically distributes the workload to all active compute resources. The following sections describe the individual components of Dopia in more detail.

5 Feature Extraction and Performance Modeling

This section describes Dopia's internal code analysis tool and the machine learning model (ML) used to predict the optimal thread-level parallelism.

5.1 Static Code Analysis and Feature Extraction

Dopia's code analysis tool is derived from the Eigen Compiler Suite (ECS) [27]. ECS is a lightweight, fast, and self-contained compiler development toolchain that allows us to implement the features of interest with small effort. The compiler frontend performs lexical, syntactic, and semantic analysis and creates an abstract syntax tree (AST) of the kernel code. The analysis backend traverses the AST and collects statistics about each loop nest's number and types of memory and arithmetical operations. The aggregated values of all loop nests constitute the input features for the ML model. The compiler currently assumes OpenCL 1.2 but can be adapted to current OpenCL versions.

As discussed in Section 3, the limited memory bandwidth is the main cause of performance degradation on integrated architectures. The memory bandwidth utilization is strongly correlated with the memory access rate and the memory access pattern [7, 34]. For GPUs, the type of memory accesses issued by neighboring PEs determines whether memory accesses can be coalesced, which significantly affects the number of memory accesses issued by the GPU [1].

Table 1. Model features.

Source	Type	Feature
code	mem op	***#mem_constant***: number of memory operations to a constant address.
code	mem op	***#mem_continuous***: number of memory operations to a continuous address.
code	mem op	***#mem_stride***: number of memory operations with a constant stride.
code	mem op	***#mem_random***: number of memory operations with a random offset.
code	arith op	***#arith_int***: number of arithmetic add/mul/div operations on integer data.
code	arith op	***#arith_float***: number of arithmetic add/mul/div/special operations with floating-point data.
input	program	***work_dim***: workload dimension.
input	data	***global_size***: total number of work-items.
input	data	***local_size***: number of work-items per work-group.
param	config	***CPU_util***: normalized number of active CPU cores.
param	config	***GPU_util***: normalized number of active GPU cores.

Dopia's code analyzer collects statistics about the following types of operations (Table 1).

Memory operations. All operations accessing memory are counted. Memory operations inside loops are classified into *constant*, *continuous*, *stride*, and *random* based on their memory address pattern. As an example, consider the following kernel with a nested loop:

```
1: for (int i = 0; i < N; i ++)
2:   for (int j = 0; j < M; j ++)
3:     D[i][j] = A[i][j] + B[j][i] +
4:               C[c1] + C[B[[j][i]]];
```

Assuming a row-major data layout, the load `A[i][j]` is classified as *mem_continuous* because the accessed memory locations are continuous within the loop. The access to `B[j][i]` on line 3, on the other hand, represents an access with a constant stride and is classified as a *mem_stride*. The access `C[c1]` repeatedly accesses the same memory location and is categorized as *mem_constant*. The indirect access to array `C`, `C[B[j][i]]`, is classified as *mem_random*, while `B[j][i]` on line 4 is an access with a constant stride (*mem_stride*). The write operation to `D[i][j]` constitutes a continuous memory operation, leading to the following extracted memory-related features: *#mem_constant = 1*, *#mem_continuous = 2*, *#mem_stride = 2*, and *#mem_random = 1*.

Arithmetic operations. The ratio between the number of memory and arithmetic operations affects the memory access rate, hence, all arithmetic operations in a kernel are considered as well. Similar to the accessed address pattern of memory operations, the type of an arithmetic operation is an important feature to predict performance. This is, e.g., because GPUs can process floating-point data more efficiently than CPUs. Dopia's code analyzer distinguishes between integer and floating-point arithmetic operations.

Table 2. Parameters of the synthetic workload α**mat**β**d** γ**c** δ**T** ϵ**R** θ**C** *dim dtype*

Param	Description
α	Number of matrices to add. "`3mat`" computes: $\forall_i$ `C[i] = A[i] + B[i] + C[i]`
β	Dimension of the matrices. "`3mat2d`" computes: $\forall_{i,j}$ `C[i][j] = A[i][j] + B[i][j]+ C[i][j]`
γ	Number of computational operations. For example, "`2mat2d2c`" yields: `C[i][j] = c1*c2*A[i][j] + c1*c2*B[i][j];`
δ	Number of matrices with transposed memory accesses. "`2mat2d2c1T`" computes: `C[i][j] = c1*c2*A[i][j] + c1*c2*B[j][i];`
ϵ	Number of matrices with randomized memory accesses. "`2mat2d2c1R`" computes: `C[i][j] = c1*c2*A[i][j] + c1*c2*B[D[j]];`
θ	Number of matrices for which memory accesses are constant. "`2mat2d2c1C`" computes: `C[i][j] = c1*c2*A[i][j] + c1*c2*B[c3];`
dim	Work-item dimension of the kernel.
dtype	Data type of matrices and operations.

Input data. The input data's dimensionality and the number and composition of the work-groups constitute other important features for performance estimation. These features are not available at compile-time but only when the kernel is submitted for execution using the `clEnqueueNDRangeKernel` API. Dopia extracts the features *work_dim*, determining the number of dimensions of the kernel, and *global/local_size* that represent the size of the workload (the total number of work-items) and the granularity of a work unit (the number of work-items in a work-group).

5.2 Performance Modeling

Dopia relies on an offline pre-trained ML model to predict the best thread-level parallelism for the CPU and GPU. The model takes as its input a feature vector comprising the eleven features listed in Table 1 and outputs the expected normalized performance of the configuration. The advantage of an ML-based approach is its ability to detect the often inconspicuous correlations between the input features and the runtime of a kernel. In addition, performance models for different architectures can be generated automatically for different architectures. Compared to other optimization approaches such as binary search or gradient descent, an ML-based approach is more computationally efficient because it does not require any profiling runs at runtime.

```
__kernel void 2mat3d(__global float* A,
                     __global float* B,
                     __global float* C,
                     int NZ, int NY, int NX)
{
 1: int z = get_global_id(0);
 2: if (z < NZ) {
 3:   for (int y = 0; y < NY; y++) {
 4:     for (int x = 0; x < NX; x++) {
 5:       int idx = z*(NY*NX) + y*NX + x;
 6:       C[idx] = A[idx] + B[idx];
 7: } } }
}
```

```
__kernel void 2mat3d(__global float* A,
                     __global float* B,
                     __global float* C,
                     int NZ, int NY, in NX,
                     int dop_gpu_mod, int dop_gpu_alloc)
{
10: __local int local_worklist[1];
11: if (get_local_id(0) == 0) local_worklist[0] = 0;
12: barrier(CLK_LOCAL_MEM_FENCE);

13: if (get_local_id(0) % dop_gpu_mod < dop_gpu_alloc)) {
14:   for (int dynamic_work = atomic_inc(local_worklist);
          dynamic_work < get_local_size(0);
          dynamic_work = atomic_int(local_worklist))
15:   {
 1:     int z = get_global_id(0);
16:     int z = get_group_id(0)*get_local_size(0) +
                    get_global_offset(0) + dynamic_work;
17:     if (z < NZ) {
18:       for (int y = 0; y < NY; y++) {
19:         for (int x = 0; x < NX; x++) {
20:           int idx = z*(NY*NX) + y*NX + x;
21:           C[idx] = A[idx] + B[idx];
22:     } } }
23:   }
24: }
}
```

Figure 5. Code transformation of a 1-dimensional kernel for malleable execution on a GPU.[1]

To obtain sufficient training data for the ML model, we execute a parameterizable synthetic workload with a total of eight parameters. The basic operation of the workload is an addition of α matrices with a dimension of β. The γ parameter allows control over the computational intensity of the kernel. Parameters δ, ϵ, and θ control the generated memory access patterns by specifying the number of matrices that are accessed with stride, indirect, and constant accesses, respectively. The seventh parameter *dim* defines the dimension of the input data, and the eighth parameter *dtype* determines the data type of the matrices and thus the arithmetic operations. Table 2 lists the parameters and describes their effect in more detail.

By varying the code and input parameters, we generate a total of 1,224 synthetic workloads (Table 4). Each workload is executed with all possible combinations of thread counts on the CPU and the GPU, and its performance is recorded. Gathering this training data for a specific integrated architecture takes a few hours. For the prediction, any suitable machine learning model can be used. Dopia employs a DecisionTree model because it offers good prediction accuracy at a low inference overhead. Model training is performed with the `scikit-learn` [30] library. The generated decision tree is converted to C code and invoked by Dopia for at-runtime model inference. A detailed analysis of different machine learning techniques is given in Section 9.2.

```
__kernel void 2mat3d(__global float* A,
                     __global float* B,
                     __global float* C,
                     int NZ, int NY, int NX)
{
 1: int z = get_global_id(0);
 2: int y = get_global_id(1);
 3: if ((z < NZ) && (y < NY)) {
 4:   for (int x = 0; x < NX; x++) {
 5:     int idx = z*(NY*NX) + y*NX + x;
 6:     C[idx] = A[idx] + B[idx];
 7: } }
}
```

```
__kernel void 2mat3d(__global float* A,
                     __global float* B,
                     __global float* C,
                     int NZ, int NY, in NX,
                     int dop_gpu_mod, int dop_gpu_alloc)
{
10: __local int local_worklist[1];
11: if (get_local_id(0) == 0) local_worklist[0] = 0;
12: barrier(CLK_LOCAL_MEM_FENCE);

13: if (get_local_id(0) % dop_gpu_mod < dop_gpu_alloc)) {
14:   for (int dynamic_work = atomic_inc(local_worklist);
          dynamic_work < get_local_size(0)*get_local_size(1);
          dynamic_work = atomic_int(local_worklist))
15:   {
 1:     int z = get_global_id(0);
 2:     int y = get_global_id(1);
16:     int z = get_group_id(0)*get_local_size(0) +
                    get_global_offset(0) +
                    dynamic_work / get_local_size(1);
17:     int y = get_group_id(1)*get_local_size(1) +
                    get_global_offset(1) +
                    dynamic_work % get_local_size(1);
18:     if ((z < NZ) && (y < NY)) {
19:       for (int x = 0; x < NX; x++) {
20:         int idx = z*(NY*NX) + y*NX + x;
21:         C[idx] = A[idx] + B[idx];
22:     } }
23:   }
24: }
}
```

Figure 6. Code transformation of a 2-dimensional kernel for malleable execution on a GPU.[1]

6 Malleable Code Generation

Dopia automatically generates CPU code and a malleable GPU version of the original OpenCL kernel. The process is illustrated with the `2mat3d` kernel that computes the sum of two three-dimensional matrices. Figures 5 and 6 show the original kernel in a 1/2-dimensional workspace and the transformed malleable GPU kernels. Figure 7 shows the generated CPU code for the 1-dimensional workspace. In the 1-dimensional workspace, one work-item computes a given z plane (Figure 5, line 1) by iterating through the y and x dimensions (lines 3–7). The z index is set to the work-item's index in the global work-item space. In the 2-dimensional workspace, z and y are given by the position of the work-item in the global work-item space (Figure 6, lines 1–2), and the kernel only loops over x (lines 4–6).

The lower parts of Figures 5 and 6 show the code after the automatic code transformation that renders the kernels malleable. Code changes are marked with **bold** and ~~strikethrough~~ text. The basic idea is to throttle parallelism by disabling specific cores from processing work-items. Within a work-group, work-items are allocated linearly to the PEs of

[1]The functions `get_local_id()`, `get_local_size()`, `get_group_id()`, and `get_global_offset()` are OpenCL API functions [21] used to compute a work-item's index.

```
void 2mat3d_CPU(float* A, float* B, float* C,
                int NZ, int NY, int NX,
                size_t* global_size, size_t* local_size,
                std::atomic_int worklist, size_t num_wgs)
{
10:  for (size_t wg_id = worklist->fetch_add(1);
          wg_id < num_wgs;
          wg_id = worklist->fetch_add(1))
11:  {
12:    for (size_t global_id = wg_id * local_size[0];
           global_id < wg_id * local_size[0]+local_size[0];
           global_id++)
13:    {
14:      int z = (int)global_id;
 1:      int z = get_global_id(0);
15:      if (z < NZ) {
16:        for (int y = 0; y < NY; y++) {
17:          for (int x = 0; x < NX; x++) {
18:            int idx = z*(NY*NX) + y*NX + x;
19:            C[idx] = A[idx] + B[idx];
20:      } } }
21:    }
22:  }
}
```

Figure 7. Generated CPU code for the original OpenCL kernel in Figure 5.

a CU (Figure 2). The malleable GPU kernels retrieve a work-item's local index relative to its work-group with `get_local_id()`. The two parameters `dop_gpu_mod` and `dop_gpu_alloc` control the degree of parallelism: only PEs whose initially assigned work-item local index modulo `dop_gpu_mod` is smaller than `dop_gpu_alloc` are allowed to process work. Since not all PEs compute their assigned work-items, Dopia employs an atomic CU-local worklist to process all work-items of a work-group in a loop.

The throttling code is implemented in line 13 of the modified kernels (Figures 5 and 6). With a work-group size = 16, $dop\_gpu\_mod = 3$, and $dop\_gpu\_alloc = 1$, e.g., the GPU scheduler launches 16 threads per CU, however, only the threads assigned work-items 0, 3, 6, 9, 12, and 15 are allowed to proceed; all other threads exit immediately.

The work-item distribution is coordinated with a CU-local `local_worklist` initialized by thread (PE) 0 (lines 10–12 in the modified kernels). All active threads atomically obtain the next work-item id from the worklist (lines 13–14) until all work-items of the work-group have been processed. The indices z and y (in the 2-dimensional workspace) identifying the work-item are computed explicitly using OpenCL API calls (lines 16 and 16–17 for the one- and two-dimensional workspace, respectively).

Figure 7 shows code generated for the CPU and the 1-dimensional workspace. The function `2mat3d_CPU` executes one work-group on one core. The work-items of the work-group are processed in sequence. Dopia's runtime (Section 7) maintains a (CPU-side) atomic worklist that is accessed by all active CPU cores to fetch their next work-group (line 10). The global ID of the work-item (the z plane) is computed from the dimensions of the OpenCL workload (line 12).

The presented software-based technique is applicable to general OpenCL data-parallel applications and integrated CPU-GPU architectures that only support locally shared atomic operations (available in OpenCL 1.2) since it does not rely on CPU-GPU (global) atomic support.

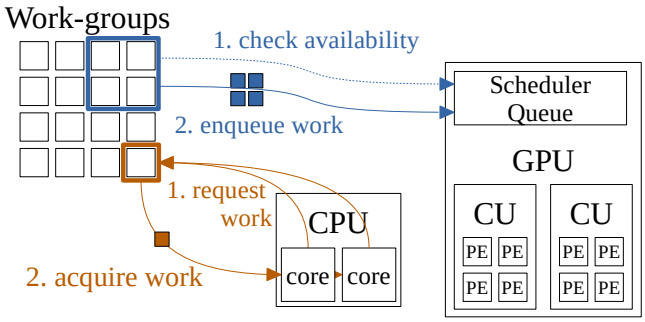


Figure 8. Workload distribution scheme in Dopia.

Algorithm 1 Runtime dynamic workload management

```
1: function ExecuteKernel(target_kernel, num_wgs, ...)
2:   # evaluate ML model
3:   dop_cpu, dop_gpu_mod, dop_gpu_alloc =
4:                 eval_model(target_kernel, num_wgs, ...))
5:   # prepare worklist and start CPU threads
6:   atomic wglist = 0
7:   for each cpucore ∈ dop_cpu do
8:     cpucore.LaunchThread(target_kernel_CPU, ..., wglist)
9:   # push chunks of work to the GPU
10:  chunksize = (num_wgs/10)
11:  repeat
12:    start = atomic_fetch_add(wglist, chunksize)
13:    gpu.EnqueueKernel(target_kernel_GPU, ...,
14:                      start, min(chunksize, num_wgs − start),
15:                      dop_gpu_mod, dop_gpu_alloc)
16:    gpu.WaitForFinish()
17:  until wglist ≥ num_wgs
```

7 Runtime Management

When a kernel is submitted for execution, Dopia combines the features obtained from static code analysis with the features only available at runtime: the dimension of the workload, the total number of work-items, and the number of work-items per work-group (refer to Table 1 for details). Although different techniques are possible, the static features are passed to the runtime via global constants in the generated CPU kernel code in the current implementation.

Dopia's ML model is evaluated for different CPU and GPU core allocations to find the best thread-level parallelism for the given kernel. The core configuration of the predicted minimal kernel runtime determines the CPU and GPU core configuration with which the kernel is executed.

Since the GPU and the CPU constitute two independent OpenCL devices, the OpenCL runtime does not provide workload distribution out-of-the-box. A workload can be distributed statically or dynamically. Predicting a static workload distribution that is well-balanced for an a priori unknown kernel is a complex problem [6, 24] and outside the scope of this paper. Dopia employs a dynamic approach that assigns the workload to the two compute devices on demand.

Figure 8 illustrates Dopia's dynamic workload distribution technique. Since Intel integrated architectures do not support

CPU-GPU global atomic operations, Dopia employs a pull-based approach for CPU threads and a push-based scheme for the GPU. Algorithm 1 shows the pseudo-code of Dopia's runtime manager. It first determines the expected best degree of parallelism for the CPU and GPU by evaluating the ML model for the given parameters (lines 2–4). The runtime manager then creates an atomic worklist to keep track of the next yet unprocessed work-group's index ($0 \leq worklist < num\_wg$) (line 6). Then, `dop_cpu` CPU threads are created to execute the generated CPU kernel code as illustrated by Figure 7 (lines 7–9). All threads have access to the atomic worklist and fetch one work-group at a time for processing until all work-groups have been processed. The runtime manager then starts pushing chunks of work to the GPU for processing. It repeatedly extracts a range of work-groups from the worklist (line 13) and assigns them to the GPU for processing (lines 14–16) using the generated malleable GPU kernel and the predicted best degree of parallelism on the GPU. After the GPU has finished processing the kernel with the assigned work-groups, the process repeats until all work-groups have been processed (lines 17–18).

In the current implementation of Dopia, the GPU's share of allocated work-groups is set to $1/10^{th}$ of all work-groups (line 11). This value was empirically found to minimize load imbalance and dispatch overhead. More elaborate work-group assignments such dynamic or application-specific work chunks or optimizations for systems that support global atomic operations (and can thus use a pull-based approach on the GPU) are left for future work.

It is also worth noticing that an ML model could be used to predict the static partitioning without relying on dynamic workload distribution. However, increasing the prediction domain would require more training data and increase the modeling overhead.

8 Experimental Setup

8.1 Environment

We have implemented Dopia for the AMD and Intel OpenCL runtime and evaluated it on an AMD A10-7850K APU (Kaveri) system [2] and an Intel Skylake i7-6700 processor system [11]. The AMD system combines a Streamroller-based quad-core CPU running at 3.7/4.0GHz (base/turbo) and a Graphics Core Next (GCN)-based GPU comprising eight CUs clocked at 720MHz with 64 PEs each (512 PEs in total). The quad-core Intel system (eight threads; 3.4/4.0GHz) integrates an Intel Gen9 HD Graphics NEO GPU clocked at 350/1150Mhz with 24 CU containing 32 PEs each (768 PEs in total).

8.2 Benchmark Scenario

Managing the degree of parallelism. Dopia considers five different levels of CPU parallelism and nine levels on the GPU. On the CPU, 0, 25, 50, 75, or 100% of all CPU resources can be activated, whereas the step size on the GPU is $1/8^{th}$, i.e., 0, 12.5, 25, …, 87.5, or 100% of all GPU resources are activated. This allows Dopia to select one of 44 possible configurations ($5 \times 9 - 1 = 44$; the configuration CPU 0, GPU 0 is excluded for obvious reasons). Table 3 lists the concrete values for the evaluated systems.

Table 3. DoP configurations on the evaluated systems.

System	CPU configuration	GPU configuration
AMD Kaveri	0, 1, 2, 3, 4 cores	0, 64, 128, …, 512 PEs
Intel Skylake	0, 2, 4, 6, 8 cores	0, 96, 192, …, 768 PEs

Model training data set. The performance model of Dopia is trained with a data set composed of data obtained from the parameterizable synthetic workload and fourteen real-world OpenCL kernels. The parameterizable synthetic workload (Section 5) is evaluated for 17 distinct memory access patterns in 72 configurations by varying the data type, workload dimension, the number of computational operations, the size of the matrices, and the size of a work-group, yielding a total of $17 \times 72 = 1,224$ data points. The real-world OpenCL kernels include twelve data-intensive OpenCL kernels from Polybench [15], `2DCONV`, `ATAX`, `BICG`, `FDTD`, `GEMM`, `GESUMMV`, `MVT` and `SYR2K`, plus a sparse-matrix and vector multiplication (`SpMV`) kernel using the compressed sparse row (CSR) format and the iterative PageRank kernel (`PageRank`) [4]. The real-world workloads are run with two different work-group organizations. Table 4 lists the evaluated configurations. The data set is generated by executing all workloads for all 44 core allocations, yielding 54,472 data points.

8.3 Comparisons

Dopia's dynamic workload distribution technique is used to compare the following resource allocation configurations:

- `CPU` processes a workload on all CPU cores. Work-items are equally divided and statically assigned.
- `GPU` processes a workload only on the GPU. using all PEs to run the kernel.
- `ALL` employs all CPU and GPU resources for collaborative execution.
- `Exhaustive` represents the DoP configuration that yields the minimal runtime from all possible 44 allocations. We employ an (unrealistic) perfect oracle that is able select this configuration without any overhead. In reality, the configuration is found through an exhaustive search over the entire parameter space.
- `Dopia` as described in this paper with automatic DoP selection based on an ML model and dynamic collaborative execution on CPU and GPU.

The different configurations are evaluated by the execution time of the kernel. This means, in particular, that all runtime overhead of Dopia (evaluation of the ML model and workload distribution) is included in the results.

Table 4. Model training data set.

1,224 parameterizable workloads (see Table 2 for notation).
17 memory access patterns (1mat3d, 1mat3d1R, 1mat3d1T, 1mat3d1C, 1mat3d1C1R, 1mat3d1C1T, 2mat3d, 2mat3d1R, 2mat3d1T, 2mat3d1R1T, 2mat3d1C, 2mat3d1C1R, 2mat3d1C1T, 2mat3d1C1R1T, 1mat4d, 1mat4d1R, 1mat4d1T) × 2 data types (float, integer) × 2 dimensions (1, 2) × 3 computational intensities ($\gamma = 0, 2, 4$) × 3 matrix sizes (16384,32768,65536) × 2 work-group sizes (64, 256 work-items per work-group)

14 real-world OpenCL kernels.

Benchmark	Input: Problem size	Input: Work-group size	Description
2DCONV	8,192	8x8, 16x16	2D Convolution
ATAX1–2	16,384	64, 256	Matrix transpose and vector multiply kernels 1–2
BICG1–2	16,384	64, 256	BiCG sub kernels 1–2
FDTD1–3	16,384	8x8, 16x16	2-D finite different time domain kernels 1–3
GESUMMV	16,384	64, 256	Scalar, vector, matrix multiply
MVT1–2	16,384	64, 256	Matrix vector product and transpose kernels 1–2
SYR2K	1,024	8x8, 16x16	Symmetric rant-2k operations
PageRank	16,384	64, 256	PageRank algorithm [4]
SpMV	16,384	64, 256	SpMV with the CSR format

9 Evaluation

In this section, we first assess the performance of Dopia's dynamic workload distribution and discuss different ML modeling approaches in terms of accuracy and overhead. The section concludes with an evaluation of the Dopia's performance for the parameterizable workloads and the real-world OpenCL workloads.

9.1 Workload Distribution

Prior to evaluating the benefits of Dopia by selecting the proper degree of parallelism, we investigate whether the runtime overhead of the dynamic approach is acceptable compared to static workload distribution. Figure 9 plots the normalized execution time of CPU-only and GPU-only execution, best `static` (no dynamic dispatch overhead), and Dopia's `dynamic` workload distribution technique evaluated with the real-world OpenCL kernels and varying input data sizes with 50 different workloads. `Static` and `dynamic` use all available compute resources. The best `static` configuration is found by evaluating 19 different static workload partitionings that assign from 5:95, 10:90, ..., 90:10, to 95:5 percent of the workload to the CPU and GPU, respectively.

We expect the runtime overhead of dynamic workload distribution to slow down execution; however, the results

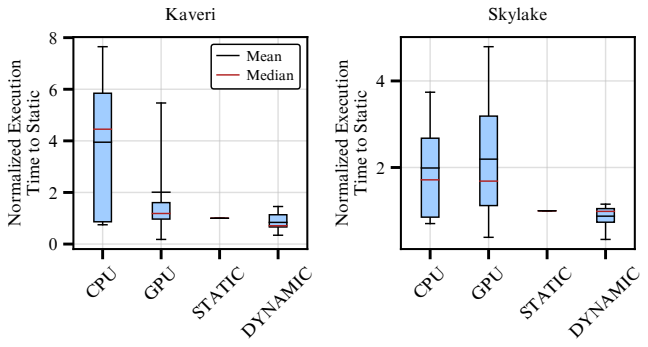
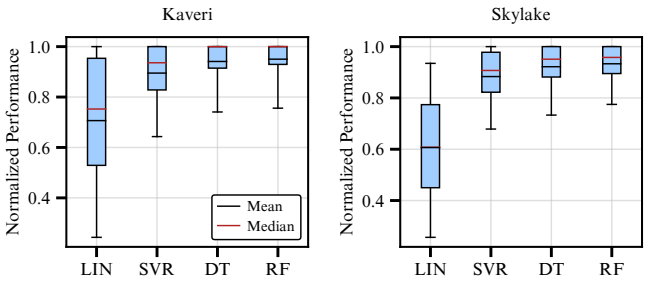


Figure 9. Normalized execution time of dynamic workload distribution compared to CPU, GPU, and static workload distribution. The black and red lines indicate the mean and median values, respectively. The boxes represent the 25th and 75th percentile (lower/upper end), and the bottom/top whiskers show the 5th and 95th percentile of the distribution.

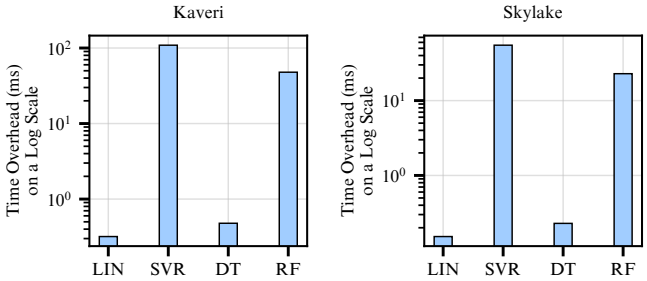
in Figure 9 show that the dynamic workload distribution technique achieves similar or even better performance than `static`. The reason for this somewhat counter-intuitive result is that the dynamic approach manages the workload distribution at a finer-grained level than the static approach with its 5% step size. In other words, the runtime overhead of dynamic workload distribution is small enough to outperform a coarser-grained static approach. Compared to CPU or GPU only execution, Figure 9 reveals that exclusive use of only one type of cores results in a significantly higher average execution time and that co-execution is required to achieve maximum performance.

9.2 ML Model Accuracy and Overhead

Different ML techniques can be used to build an ML performance model for Dopia. To choose a technique, we use the profiling data of parameterizable workloads and compare the accuracy and overhead of different ML techniques. Figure 10 plots model accuracy and model inference overhead of four different modeling techniques: `Lin` (Linear Regression), `SVR` (Support Vector Regression), `DT` (Decision Tree), and `RF` (Random Forest). The models are trained using all 44 core configurations of the 1,224 parameterizable workloads and evaluated with 64-fold cross-validation. Cross-validation is a commonly used approach to validate the modeling accuracy for a dataset with a limited number of data points. The dataset is randomly shuffled and divided into 64 equal-sized groups. Each group represents the test set once, while the remaining 63 are used to train the model. The reported results represent the average accuracy of all 64 models. We observe that tree-based approaches such as `DT` and `RF` outperform regression approaches such as `Lin` and `SVR` for the parameterizable workloads. Figure 10 (b) reveals that `Lin` and `DT` have a several orders of magnitude lower inference

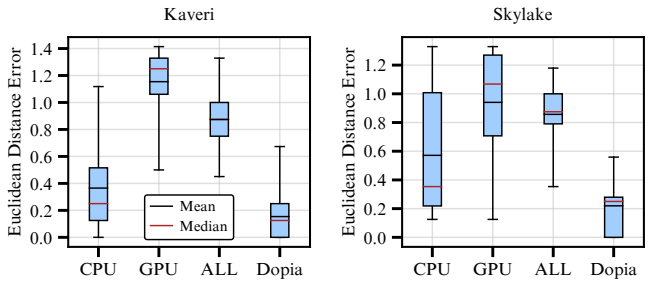


(a) Model accuracy

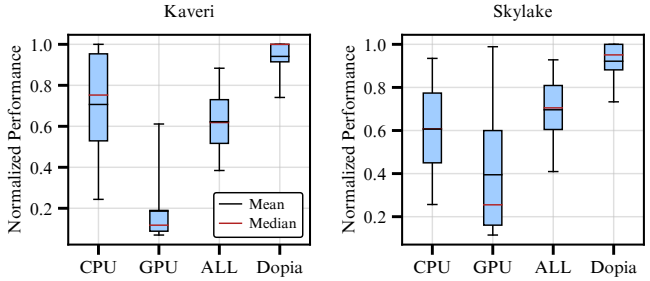


(b) Inference overhead evaluating all 44 configurations.

Figure 10. Comparing prediction accuracy and inference overheads between different ML modeling approaches.



(a) Euclidean distance error



(b) Normalized performance

Figure 11. 64-fold cross validation on 1,224 parameterizable workloads.

Table 5. Correct number of classifications of Dopia and the three static configurations for 1,224 workloads.

	CPU	GPU	ALL	Dopia
Kaveri	253	15	7	611
Skylake	27	57	19	334

overhead than SVR and RF that use complex modeling techniques. Since the kernels and input data sizes used in this work have relatively short execution times in the order of 1-2 seconds, overhead is a significant concern. Based on the high accuracy and low overhead, in this work, Dopia employs a DT-based model. We note that the modeling technique can be tuned depending on the workload types and problem sizes. In particular, if the kernel runtimes are in the order of 10 seconds or higher, more accurate models such as RF and SVR can be considered.

9.3 Parameterizable Workloads

In this section, we evaluate the performance of Dopia's ML model by counting the number of correct predictions of the model compared to the number of times the three static approaches CPU, GPU, and ALL represent the best execution mode for the parameterizable workloads. The results in Table 5 show that Dopia's model-guided approach outperforms the static approaches by a wide margin.

While the model's accuracy is not exceptionally high with 611/1224 = 50% for Kaveri and 334/1224 = 27% on Intel, we observe that minor prediction errors still lead to close-to-optimal performance, as is evident from the performance heat maps in Figure 1 and 12. For example, consider the case where the best configuration for a workload is 1 CPU thread and 512 GPU threads. Even if the model fails to predict this configuration and selects a configuration with 2 CPU threads and 512 GPU threads, this setup is still likely to achieve close-to-optimal performance. To analyze how close the predictions are to the best configurations in this two-dimensional parameter space, Figure 11 (a) plots the error using the Euclidean distance metric from the selected to the best configuration. The metric is obtained by measuring the Euclidean distance of each predicted configuration to the optimal configuration. This value is then normalized by dividing it by the longest distance in our problem space, $sqrt(1.0^2 + 1.0^2)$. In comparison to the absolute prediction accuracy in Table 5, Figure 11 (a) demonstrates that the mean Euclidean distance error of Dopia is significantly lower, with 15% on the Kaveri system and 22% on the Skylake system. Looking at tail performance, Dopia experiences an error of about 20–30% for the 75th percentile, outperforming the other configurations by a large margin.

To verify that Dopia achieves good overall performance, we compare the normalized performance of the predicted best configuration against the best-known one (Exhaustive). Figure 11 shows the relative normalized performance of the 1,224 workloads for Dopia and the three fixed partitioning approaches. Similar to the analysis of the Euclidean error, Dopia achieves 94% (Kaveri) and 92% (Skylake) of performance on

Table 6. Normalized performance of the static partitionings CPU, GPU, ALL and the best overall configuration from Figure 12 compared to Exhaustive.

Configuration	Degree of parallelism	Kaveri	Skylake
CPU	CPU 1.0, GPU 0	70.7%	60.7%
GPU	CPU 0, GPU 1.0	18.6%	39.5%
ALL	CPU 1.0, GPU 1.0	62.3%	69.6%
Best const.alloc.	CPU 1.0, GPU 0.125	82.5%	81.6%
Dopia	Driven by ML model	94.1%	92.2%

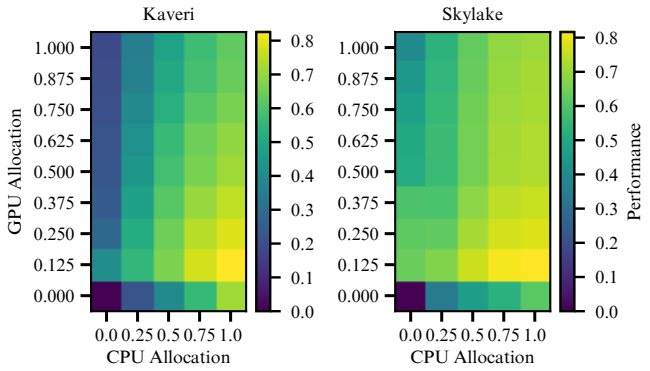


Figure 12. Normalized performance for different constant CPU-GPU thread configurations.

average compared to Exhaustive. A notable observation is that conventional co-execution using all available CPU and GPU resources performs significantly better on Intel. The reason is that the Intel i7-6700 processor provides more memory bandwidth and contains a shared last-level cache. Overall, Dopia significantly outperforms the CPU, GPU, and ALL configurations regardless of the hardware platform.

Table 6 and Figure 12 demonstrate the necessity of an automatic approach to parallelism management on integrated architectures. The heat map in Figure 12 shows the averaged normalized performance of all 1,224 parameterizable workloads for different CPU and GPU thread allocations on the AMD and the Intel system. For each kernel, the values of the cells in Figure 12 are computed by dividing the execution time of the corresponding cell's configuration by the shortest execution time. The heat map shows the average cell value over all kernels. Table 6 shows the normalized performance of the three fixed resource allocations CPU, GPU, and ALL plus the best constant configuration overall Best constant allocation. We observe that Dopia significantly improves the performance of all simple constant configurations.

9.4 Real-world workloads

This section evaluates the performance of Dopia and the other co-running approaches compared to the best obtainable performance by a perfect oracle. For the evaluation of the 14 OpenCL real-world kernels as listed in Table 4, we use the parameterizable workloads and the real-world applications for training; however, the kernel under evaluation is excluded from the training dataset. Except for the two-dimensional kernels 2DCONV, FDTD, and SYR2K, all other kernels operate on one-dimensional input data. 2DCONV uses a work-group size of 8x8, while all other kernels use a work-group size of 256. The input graph for PageRank and SpMV has 16,384 rows, and the number of elements per row in CSR format is also 16,384.

Figure 13 plots the performance of CPU, GPU, ALL, and Dopia for the 14 evaluated OpenCL workloads. While Dopia uses the DT ML modeling technique, we also provide details about the LIN, SVR, and RF modeling techniques. On average, Dopia (DT) achieves 84% of the normalized best (oracle) performance on both systems, including all runtime overhead. With SVR, the most accurate model, and disregarding the model inference overhead, Dopia would achieve 88% of the normalized best performance on both systems; however, this is not a realistic scenario. The relatively large model inference overhead of SVR compared to the real-world kernels' total execution time decreases the overall benefit of SVR-based runs to 64% and 70% on Kaveri and Skylake, respectively. This observation, however, suggests that for larger kernels, Dopia can make use of more sophisticated modeling techniques to obtain better performance.

Comparing the performance of the parameterizable workloads (Figure 11) to the real-world kernels, we observe that the overall performance trend is similar. With the real-world kernels, the second-best configuration ALL achieves an average performance of 76% on Kaveri and 75% on the Skylake system, demonstrating that selecting the proper thread-level parallelism is necessary to obtain the full processing power of integrated processors. We observe that CPU outperforms GPU for the parameterizable workloads, while for real-world kernels, GPU outperforms CPU. This is because the parameterizable workloads are designed to evaluate memory access pressure, and such workloads tend to be CPU-friendlier. On the other hand, real-world applications contain several GPU-friendly applications such as 2DCONV, FDTD1-3, and MVT2.

Overall, the results demonstrate that Dopia provides consistently good performance for workloads with different performance characteristics. Looking at individual workloads, in most cases, Dopia outperforms the other three configurations or achieves similar performance, suggesting that adjusting the number of CPU and GPU threads can improve co-running performance over the state-of-the-art resource allocation approaches [37] that select the best configuration from CPU, GPU, and ALL configurations.

As an exception to the rule, we observe that Dopia fails to select a good configuration for MVT2. This is especially true on the Skylake system where the performance of Dopia is below 40% of the maximal performance. Our analysis revealed

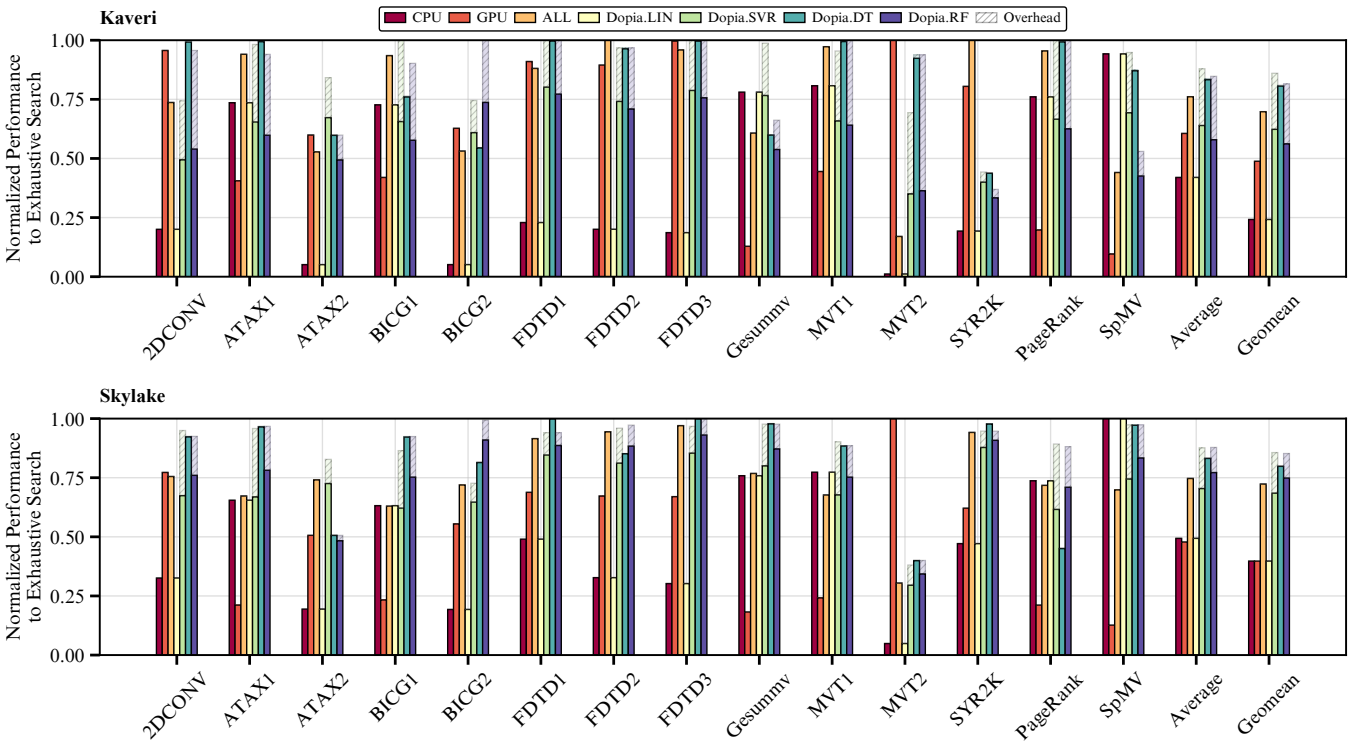


Figure 13. Performance of Dopia for 14 real-world kernels and different ML modeling techniques. The overhead bar shows the performance drop incurred by model inference.

two reasons for this result. First, the static code analysis extracts an identical feature vector despite different performance behavior for `MVT2` and `ATAX2`, suggesting room for improvement in the static code analysis and feature selection process. The other reason is that `MVT2` performs only a number of simple transpose operations on a data buffer, and the GPU can execute these operations efficiently without collaborative execution on the CPU. While the performance of most benchmarks is relatively insensitive to minor mispredictions, this is not true for `MVT2` where the small misprediction of Dopia's ML model causes a significant performance drop. This can be alleviated by more training data from GPU-friendly kernels such that the ML model can make better predictions overall.

10 Conclusion

Dopia provides an automatic and fully software-based approach for workload partitioning on integrated CPU-GPU architectures. Dopia dynamically analyzes and modifies OpenCL kernels as they are enqueued by an application for execution. An integrated code analyzer extracts performance-relevant features that are fed into a machine learning model to predict the number of CPU and GPU cores expected to yield optimal performance. Dopia prepares the OpenCL kernel for execution on integrated architectures by injecting code that facilitates dynamic load balancing. Trained and evaluated with 1,224 execution profiles of OpenCL kernels and 14 real-world OpenCL kernels, we have shown that Dopia outperforms standard workload partitioning schemes by a significant margin. The machine learning model is able to select a configuration that lies within 10–20% of the best possible configuration. Dopia is a software-only solution that does not require any hardware modification, and the ML modeling features collected from a code analysis are applicable to any processor. As integrated architectures continue to evolve and their complexity increases, such software-only approaches are likely to be of interest to a large group of users.

The Dopia software framework, including all training data, is available at https://csap.snu.ac.kr/software.

Acknowledgments

We thank the anonymous reviewers and our shepherd for the helpful feedback. This work was supported by the National Research Foundation of Korea (NRF) through grants 2015K1A3A1A14021288 and 21A20151113068 (BK21 Plus for Pioneers in Innovative Computing – Dept. of Computer Science and Engineering, SNU), and the Swiss National Science Foundation (SNF) grant IZKSZ2_162084 as part of the Korean-Swiss Science and Technology Programme (KSSTP). ICT at Seoul National University provided research facilities for this study. Younghyun Cho was previously with Seoul National University when this research was initiated.

References

[1] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[2] Dan Bouvier and Ben Sander. 2014. Applying AMD's Kaveri APU for heterogeneous computing. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. 1–42. https://doi.org/10.1109/HOTCHIPS.2014.7478810

[3] Alexander Branover, Denis Foley, and Maurice Steinman. 2012. AMD Fusion APU: Llano. *IEEE Micro* 32, 2 (March 2012), 28–37. https://doi.org/10.1109/MM.2012.2

[4] Sergey Brin and Lawrence Page. 2012. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer Networks* 56, 18 (2012), 3825 – 3833. https://doi.org/10.1016/j.comnet.2012.10.007

[5] Younghyun Cho, Camilo A. Celis Guzman, and Bernhard Egger. 2018. Maximizing System Utilization via Parallelism Management for Co-located Parallel Applications. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/3243176.3243199

[6] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R. Gross. 2018. On-the-fly Workload Partitioning for Integrated CPU/GPU Architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 21, 13 pages. https://doi.org/10.1145/3243176.3243210

[7] Younghyun Cho, Surim Oh, and Bernhard Egger. 2016. Online scalability characterization of data-parallel programs on many cores. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 191–205. https://doi.org/10.1145/2967938.2967960

[8] Younghyun Cho, Surim Oh, and Bernhard Egger. 2020. Performance Modeling of Parallel Loops on Multi-Socket Platforms using Queueing Systems. https://doi.org/10.1109/TPDS.2019.2938172

[9] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. 2008. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 250–259. https://doi.org/10.1145/1454115.1454151

[10] M. Damschen, F. Mueller, and J. Henkel. 2018. Co-Scheduling on Fused CPU-GPU Architectures With Shared Last Level Caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (Nov 2018), 2337–2347. https://doi.org/10.1109/TCAD.2018.2857042

[11] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37, 2 (Mar.-Apr. 2017), 52–62. https://doi.org/10.1109/MM.2017.38

[12] Murali Krishna Emani and Michael O'Boyle. 2015. Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 499–508.

[13] Reza Entezari-Maleki, Younghyun Cho, and Bernhard Egger. 2020. Evaluation of memory performance in NUMA architectures using Stochastic Reward Nets. *J. Parallel and Distrib. Comput.* 144 (October 2020), 172–188. https://doi.org/doi.org/10.1016/j.jpdc.2020.05.022

[14] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Victor García-Floreszx, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 43–54. https://doi.org/10.1109/ISPASS.2017.7975269

[15] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339595

[16] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 152–163. https://doi.org/10.1145/1555754.1555775

[17] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/2451116.2451158

[18] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. 2014. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/2628071.2628088

[19] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 157–166. http://dl.acm.org/citation.cfm?id=2523721.2523745

[20] Khronos Group. 2021. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. [online; accessed January 2022].

[21] Khronos OpenCL Working Group. 2012. The OpenCL Specification Version: 1.2. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf. [online; accessed January 2022].

[22] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 277–288. https://doi.org/10.1145/1941553.1941591

[23] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. 2005. Heterogeneous chip multiprocessors. *Computer* 38, 11 (Nov 2005), 32–38. https://doi.org/10.1109/MC.2005.379

[24] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. *ACM Transactions on Computer Systems* 33, 3, Article 9 (Aug. 2015), 27 pages. https://doi.org/10.1145/2798725

[25] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4, Article 69 (July 2015), 35 pages. https://doi.org/10.1145/2788396

[26] CUDA Nvidia. 2007. Compute unified device architecture programming guide. (2007).

[27] Openbrace. 2021. Eigen Compiler Suite. https://ecs.openbrace.org/. [online; accessed January 2022].

[28] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 457–467.

[29] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A System for Flexible Parallel Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2254064.2254082

[30] scikit-learn. 2018. scikit-learn - Machine Learning in Python, v0.19.2. http://scikit-learn.org/stable/. [online; accessed January 2022].

[31] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. 2016. Workload Partitioning for Accelerating Applications on

Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (Sept 2016), 2766–2780. https://doi.org/10.1109/TPDS.2015.2509972

[32] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter Mccardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581262

[33] Terrence Sych. 2014. OpenCL Device Fission for CPU Performance. https://www.codeproject.com/Articles/849911/OpenCL-Device-Fission-for-CPU-Performance. [online; accessed January 2022].

[34] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2016. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 419–431. https://doi.org/10.1109/HPCA.2016.7446083

[35] Junsung Yook and Bernhard Egger. 2021. Modeling Cache and Application Performance on Modern Shared Memory Multiprocessors. In *Proceedings of the 19th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2021)*. 1151–1158. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00158

[36] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. 2017. FinePar: Irregularity-aware Fine-grained Workload Partitioning on Integrated Architectures. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 27–38. http://dl.acm.org/citation.cfm?id=3049832.3049836

[37] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2017. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (March 2017), 905–918. https://doi.org/10.1109/TPDS.2016.2586074

[38] Qi Zhu, Bo Wu, Xipeng Shen, Kai Shen, Li Shen, and Zhiying Wang. 2017. Understanding co-run performance on CPU-GPU integrated processors: observations, insights, directions. *Frontiers of Computer Science* 11, 1 (01 Feb 2017), 130–146. https://doi.org/10.1007/s11704-016-5468-8