PySchedCL: A framework for Partitioning and Scheduling OpenCL Programs

Project-II (CS57004) report submitted to the
Indian Institute of Technology, Kharagpur
In partial fulfilment for the award of the degree

of

Master of Technology
in
Computer Science & Engineering

by

Lokesh Dokara

(12CS30017)

Under the supervision of

Asst. Prof. Soumyajit Dey

Department of Computer Science & Engineering

Indian Institute of Technology, Kharagpur

Spring Semester, 2016-17

May 2017

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: (Lokesh Dokara)

Place: Kharagpur (12CS30017)

Department of Computer Science & Engineering

Indian Institute of Technology, Kharagpur

Kharagpur - 721302, India

CERTIFICATE

This is to certify that the project report entitled "**PySchedCL: A framework for Partitioning and Scheduling OpenCL Programs**" submitted by **Lokesh Dokara** (Roll No. 12CS30017) to Indian Institute of Technology, Kharagpur towards partial fulfilment of requirements for the award of degree of Master of Technology in Computer Science & Engineering is a record of bona fide work carried out by him under my supervision and guidance during Spring Semester, 2016-17.

Asst. Prof. Soumyajit Dey,

Department of Computer Science & Engineering,

Indian Institute of Technology, Kharagpur

Date:

Abstract

Name of the student: **Lokesh Dokara** Roll No: **12CS30017**
Degree for which submitted: **Master of Technology**
Department: **Department of Computer Science & Engineering**
Thesis title: **PySchedCL: A framework for Partitioning and Scheduling OpenCL Programs**
Thesis supervisor: **Asst. Prof. Soumyajit Dey**
Month and year of thesis submission: **May 2017**

Heterogeneous computing systems consisting of multiple CPUs and GPUs are increasingly attractive as they deliver high performance at relatively low energy costs. This trend has spread to the desktop, where the high-end relies on accelerator devices for increased performance. With the rise of GPGPU (general-purpose computing on GPUs), heterogeneous computing has become increasingly prevalent and attractive for more mainstream programming. OpenCL has emerged as a standard which provides program portability by allowing the same program to execute on different types of devices. Although it provides portable functionality allowing user to program multiple different devices from a single framework, its performance will vary drastically across different components of the heterogeneous system as processors in such systems are often based on entirely different architectures.

The work reported in this thesis describes a generic scheduling framework which can partition workload of OpenCL tasks between CPU and GPU and schedule them across multiple CPUs and GPUs. This framework eliminates the need to write OpenCL host programs thus increases the ease of writing OpenCL programs.

Acknowledgements

I am deeply grateful to my supervisor **Asst. Prof. Soumyajit Dey** who gave me the opportunity to work on this project. I am thankful for his aspiring guidance, invaluably constructive friendly advice during the course of the project. I am sincerely grateful to him for sharing his truthful and illuminating views on a number of issues related to the project. I owe a lot to my teachers in the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, who have instilled in me the scientific spirit of inquiry, experimentation, observation and inference, without which I would not have been able to produce this work. Also, I am very thankful to **Mr. Anirban Ghose** involved in this project who constantly motivated me to overcome all the challenges and helped me whenever I faced any issues.

Contents

List of Figures

List of Tables

Abbreviations

GPU	**G**raphic **P**rocessing **U**nit
CPU	**C**entral **P**rocessing **U**nit
OpenCL	**Open C**omputing **L**anguage

Symbols

$\mathcal{K}$	A set of Kernel objects which represent the taskset.
$\mathcal{Q}$	Represents the *QueueSet* object, which is a set of queues containing the Kernel.
$nCPU$	number of available CPU devices.
$nGPU$	number of available GPU devices.
$\mathcal{F}$	A frontier of independent Kernel objects.

Chapter 1

Overview

1.1 Introduction

Heterogeneous computing systems consisting of multiple CPUs and GPUs are increasingly attractive as they deliver high performance at relatively low energy costs. This move to parallel system has been mirrored by the growing use of specialised accelerators such as GPUs. By having processing units with different characteristics, computation can be mapped to specialised devices that perform a specific type of task more efficiently than other devices. This trend has spread to the desktop, where the high-end relies on accelerator devices for increased performance. With the rise of GPGPU (general-purpose computing on GPUs), heterogeneous computing has become increasingly prevalent and attractive for more mainstream programming.

OpenCL has emerged as a standard which provides program portability by allowing the same program to execute on different types of devices. Although it provides portable functionality allowing user to program multiple different devices from a single framework, its performance will vary drastically across different components of the heterogeneous system as processors in such systems are often based on entirely different architectures. Now, as such systems become more mainstream, they will

move from application dedicated devices to platforms that need to support multiple concurrent user applications. Performance variability that may be manageable when the GPU is used as a dedicated acceleration device by a single application poses a problem for concurrent users. Hence there is a need to determine when and where to map different applications to best utilise the available hardware resources and achieve a higher performance.

1.2 Objective

We aim at building an intelligent scheduling framework which takes as input a set of OpenCL kernels (which may have dependencies) and distributes the workload across multiple CPUs and GPUs in a heterogeneous multicore platform. The framework relies on a Machine Learning (ML) based ML based partition prediction framework that analyses static program features of OpenCL kernels and predicts the ratio in which kernels are to be distributed across CPUs and GPUs. Given such a scheduling strategy and a set of kernels, the framework generates device specific binaries and dispatches them across multiple devices in the heterogeneous platform as per the strategy.

1.3 Contribution

- We have developed a generic scheduling framework on top of OpenCL API with the following capabilities.
  - The framework eliminates the burden of writing lengthy OpenCL Host Programs.
  - The framework can partition workload of and OpenCL task across multiple CPUs and GPUs.

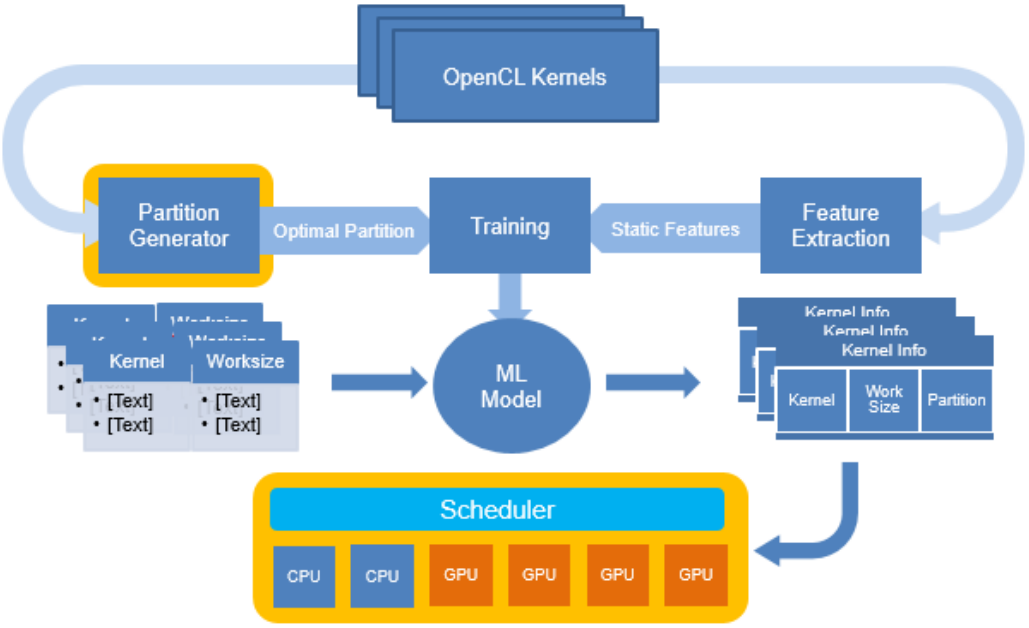

FIGURE 1.1: Overview of the project.

- Building on top of this framework we have implemented various strategies for scheduling OpenCL Tasks with and without dependencies.
- The framework and related files comprise more than 4000 lines of code.

The block diagram shows the overview of scheduling framework. This semester my work was mostly focused on improving scheduler to handle dependencies between kernels. It mainly involves

- Specifying an input format that can be used to describe dependencies between kernels and modifying framework to operate on this Directed Acyclic Graph representing kernel dependencies.
- Facilitating data transfer between kernels with dependencies.
- Maintaining a frontier of kernels that can be scheduled independently and scheduling kernels from this frontier in a partition aware manner.

- Implementing a Contraction Algorithm to reduce the data transfer overhead between OpenCL tasks.

Chapter 2

Introduction to OpenCL

OpenCL applications are data parallel programs which process multidimensional data. Every OpenCL application comprises two parts - a single threaded *host* program and a data-parallel program referred to as *kernel* that describes the computation of a single work-item. During program execution, a user-specified number of work-items is launched to execute in parallel. These work-items are organized in a multi-dimensional grid and subsets of work-items are grouped together to form work-groups, which allow work-items to interact. Each work-item can query its position in the grid by calling certain built-in functions from within the kernel code.

Despite OpenCL's focus on data-parallelism, task-parallelism is also supported in the framework to allow execution on multiple devices, e.g. multiple GPUs or CPUs and GPUs. For each device, the user can create a command queue to which (data-parallel) tasks can be submitted. This not only allows the user to execute different tasks in parallel, but also enables decomposition of data-parallel tasks into sub-tasks that are distributed across multiple devices. Because OpenCL supports a variety of processing devices, all this can be achieved with just a single implementation of each task. OpenCL's memory model reflects the memory hierarchy on graphics cards. There is a global memory that is accessible by all work-items. There is also a small local memory for each work-group that can only be accessed by work-items

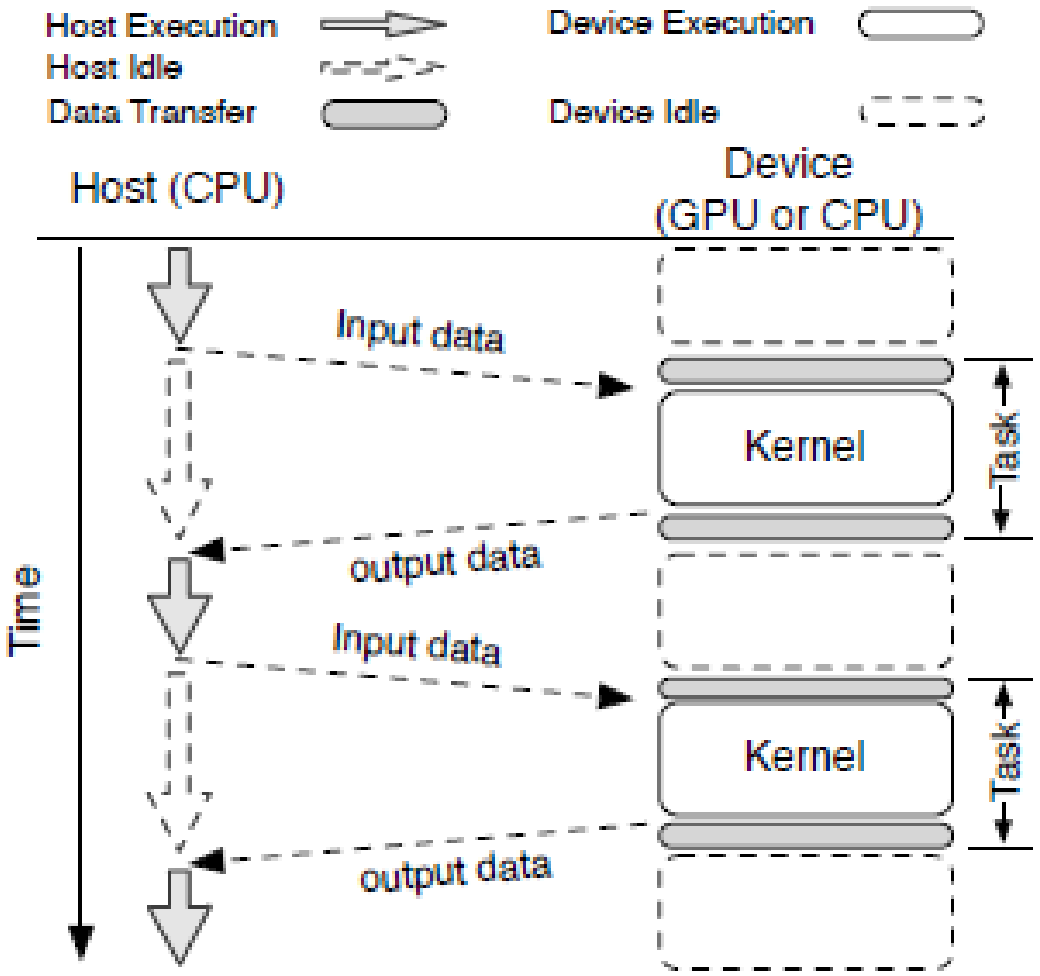

FIGURE 2.1: OpenCL Runtime.

from that particular work-group. Additionally, there is a constant memory which is read-only and can be used to store look-up tables, etc. This memory model is general enough to be mapped to many devices. Some processing devices, for example GPUs, have a global memory that is separate from the computer's main memory. In this case, any data needs to be copied to the device and back to main memory before and after task execution, and can be a considerable overhead.

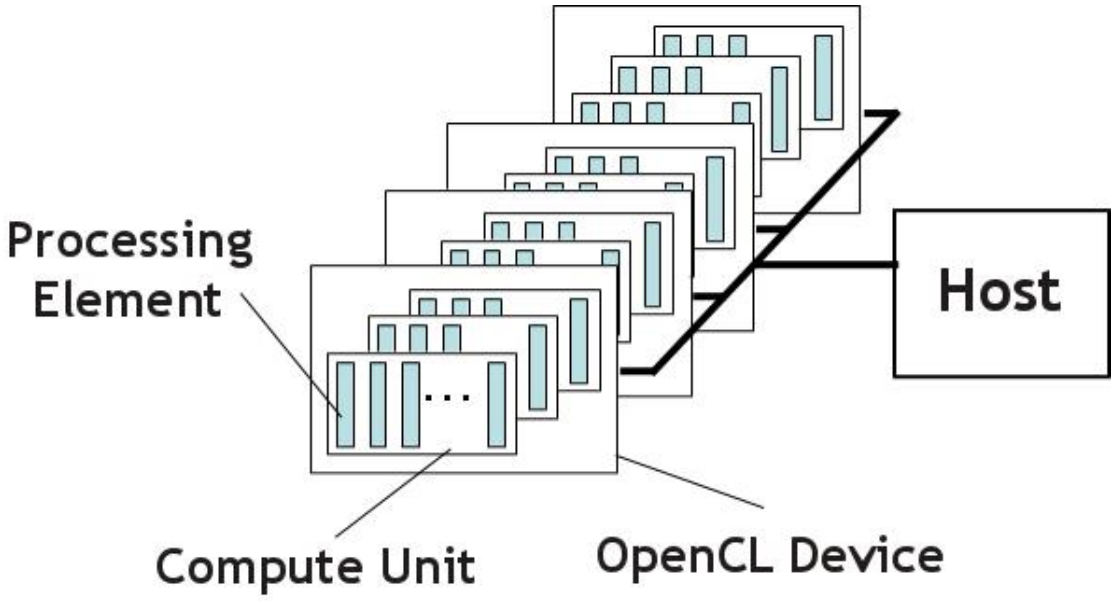

FIGURE 2.2: OpenCL Platform Model.

2.1 Architecture

An OpenCL system consists of a host and one or more OpenCL devices. A single OpenCL device typically consists of several compute units, which in turn comprise multiple processing elements (PEs). A single kernel execution can run on all or many of the PEs in parallel.

2.1.1 Memory Model

In OpenCL, five types of memory are distinguished:

- Host Memory: The host memory is the regular memory of the host program. An OpenCL kernel can not access it directly.
- Global Memory : This is the working memory of the OpenCL kernel. Each instance of a kernel has random access to the entire range.
- Constant Memory : The constant memory is different from the global memory that the kernel instances read this memory only, but can not change.

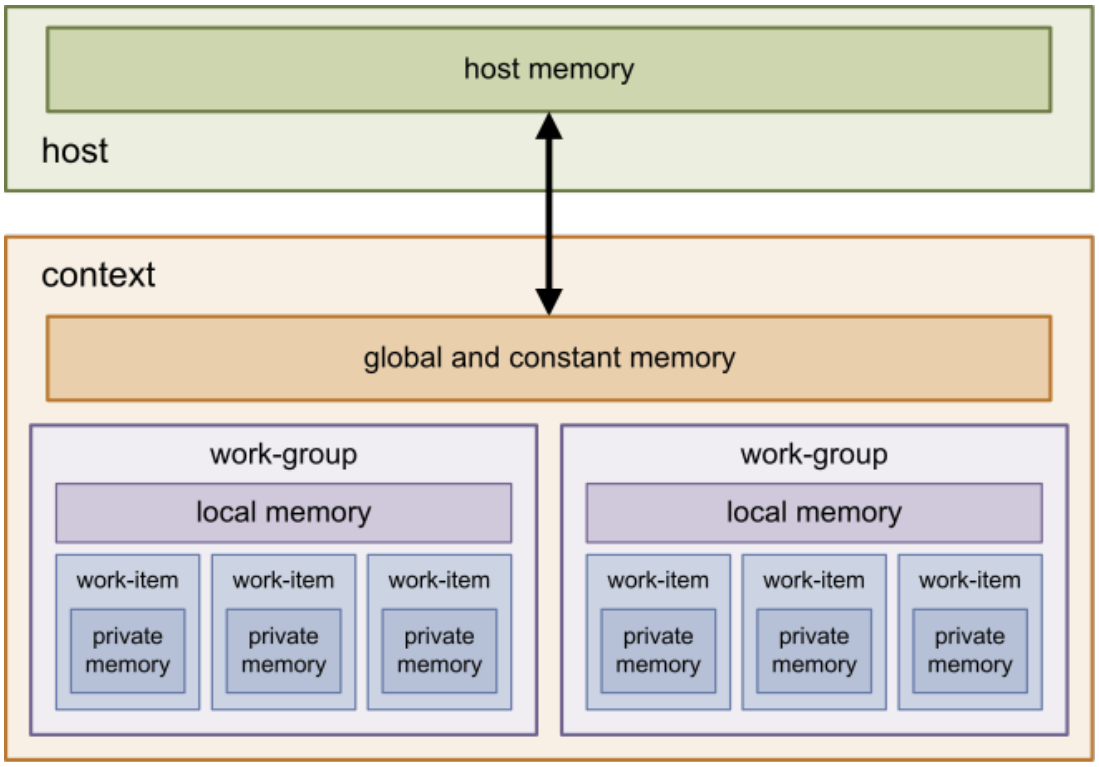

FIGURE 2.3: OpenCL Memory Model.

- Local Memory : A group of kernel instances has random access to a small area of local memory (typically 16 kB). Each group has its own area to which only members can access.
- Private Memory : This memory is a kernel Instance reserved. Other kernel instances and the master program can not access the contents of this memory.

2.2 The OpenCL Runtime API

Here are some most frequently used functions in OpenCL Runtime API:

- **clGetPlatformIDs** - Obtain the list of platforms available.
- **clGetDeviceIDs** - Obtain the list of devices available on a platform.

- **clCreateContext** - Creates an OpenCL context for the obtained devices.
- **clCreateCommandQueue** - Creates a command-queue on a specific device.
- **clCreateProgramWithSource** - Creates a program object for a context, and loads the kernel source code specified by the text strings in the strings array into the program object.
- **clBuildProgram** - Builds (compiles and links) a program executable from the program source or binary.
- **clCreateKernel** - Creates a kernel object given a program and the kernel name.
- **clSetKernelArg** - Used to set the argument value for a specific argument of a kernel.
- **clCreateBuffer** - Creates a buffer object in a context.
- **clEnqueueWriteBuffer** - Enqueues commands to write to a buffer object from host memory in a given command queue.
- **clEnqueueNDRangeKernel** - Enqueues a command to execute a kernel on a device given work dimension, global work size and global work offset.
- **clEnqueueReadBuffer** - Enqueues commands to read from a buffer object to host memory in a given command queue.

Chapter 3

Partitioning and Scheduling OpenCL Tasks

3.1 Partitioning OpenCL Tasks

The performance of OpenCL programs will vary drastically across different components of the heterogeneous system i.e., some programs run faster on CPUs, some programs run faster on GPUs and some programs run faster when partitioned across CPU and GPU. Determining the right mapping for a task is crucial to achieve good performance on heterogeneous architectures. Here we illustrate this point by examining the performance of four OpenCL programs, each of which needs a different partitioning to achieve its best performance. Figure 3.1 shows the speedup of four OpenCL programs with different mapping over single-core execution on a CPU. The x-axis shows how much of the program's workload is executed on GPU device i.e. the leftmost bar shows the speedup of CPU-only execution, one bar to the right shows the execution with 90% of work on the GPU and 10% on the CPUs and so on. For the Mean Kernel, a GPU-only execution achieves by far the best performance and for Correlation Program, CPU-only execution results in the best performance. The

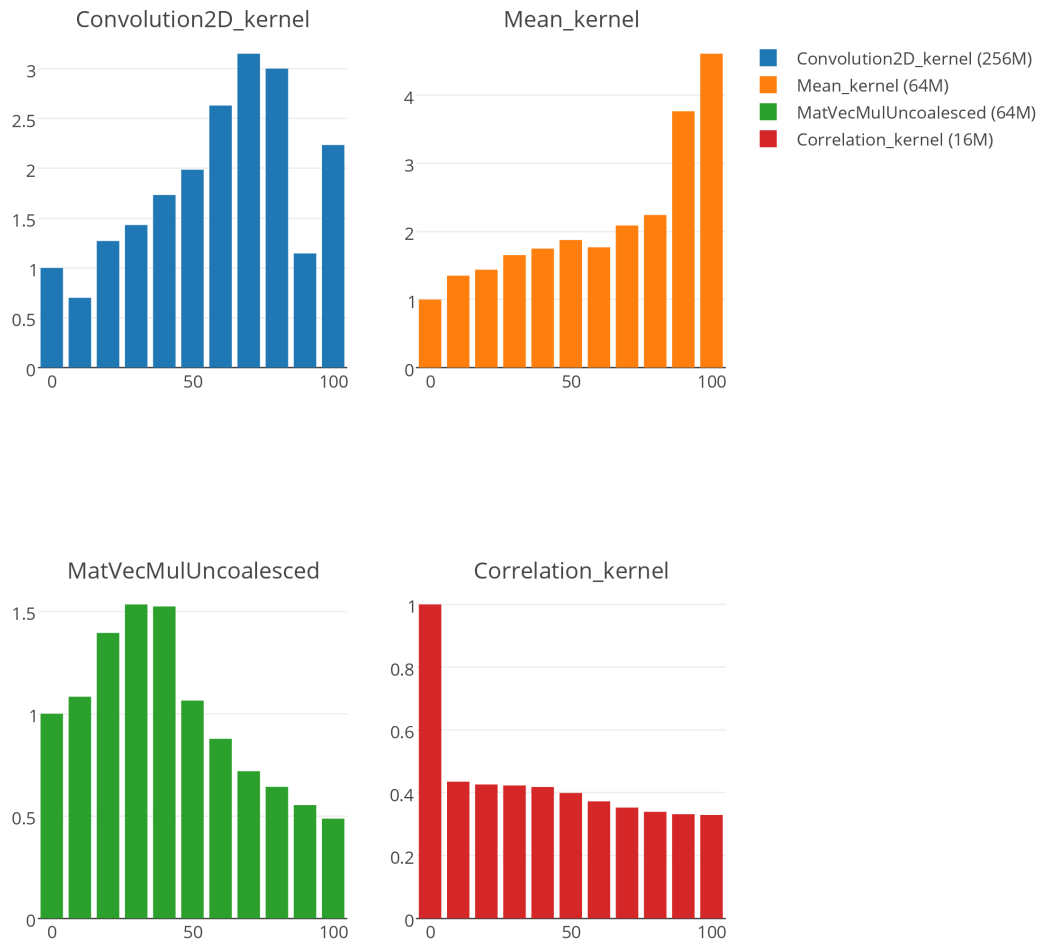

FIGURE 3.1: Work Load Partition vs Speedup for different kernels.

Uncoalesced Matrix-Vector Multiplication and 2D Convuolution kernels exhibit a totally different behaviour. Unlike the other cases, neither a GPU-only nor a CPU-only execution would achieve good performance. The highest speedup is observed with 30/70 and 70/30 partition between GPU-CPU. These behaviours can be attributed to characterstics of kernels such as computation-per-data-item and the overhead of transferring data between main memory and the GPU's memory. So it is absolutely vital to know ahead of time what the optimal mapping is.

Different programs need different mappings and for some of them making a small mistake means that large potential speedups are missed. These optimal partition values are predicted by an ML based partition prediction framework and fed as input to the Scheduling Framework.

3.2 Scheduling OpenCL Tasks

Most of the OpenCL Applications consist of more than one OpenCL kernels, and some of them are dependent on other i.e. output some kernels serves as input for others. e.g Kernels to find Correlation Matrix, Covariance Matrix. Neural Networks, Kmeans, etc.

3.2.1 Scheduling Independent Kernels

This work is concerned with the scheduling of multiple OpenCL kernel tasks on a CPU/GPU based heterogeneous platform. A kernel task is referred to as an OpenCL kernel at runtime, which includes computation and associate CPU-GPU communications. Tasks might belong to one or more than one OpenCL programs. In some cases, we partition work of a single kernel across devices to achieve better performance.

A typical scenario of OpenCL task scheduling is illustrated in figure. Here we have a task queue that is managed by a runtime scheduler. In this example, the task queue contains several OpenCL tasks submitted by four OpenCL programs, where each task can run on both the CPU and the GPU. It is therefore the runtime scheduler's responsibility to decide which device to use to run a particular task that can lead to the best overall performance (e.g. throughtput or turnaround time). We aim to develop a portable approach for efficient OpenCL multi-task scheduling and our goal is to maximize the system throughput without significantly increasing the average

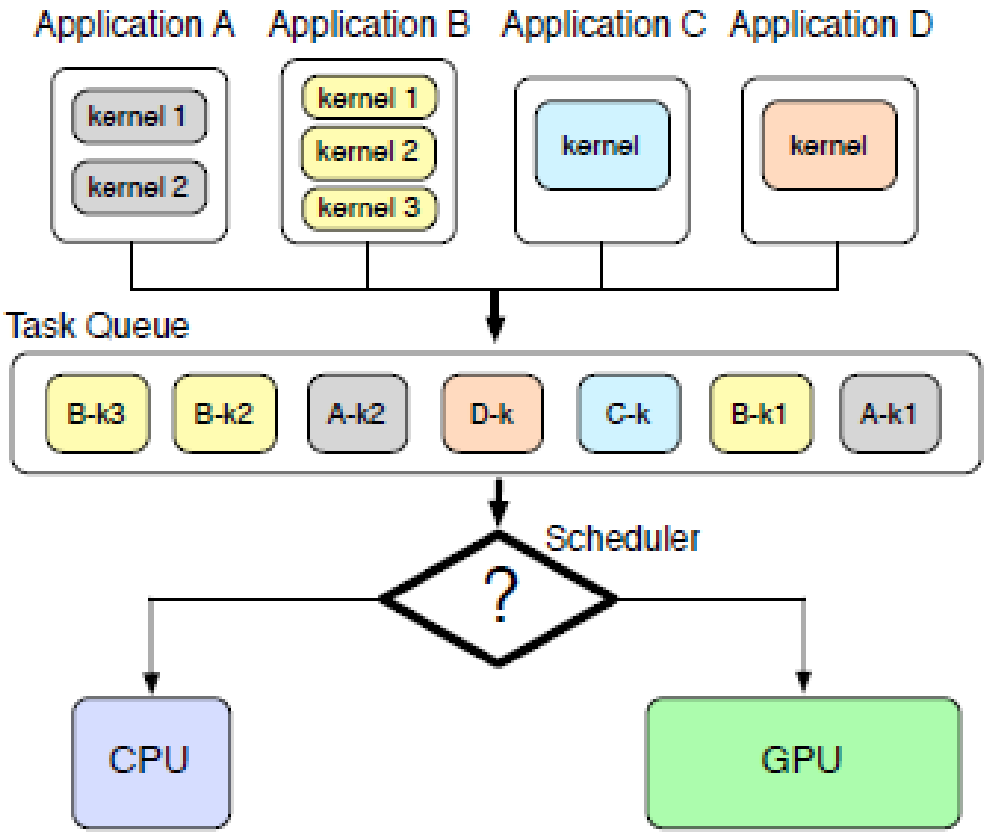

FIGURE 3.2: Typical Scheduling Scenario.

application turnaround time. The next section provides an example showing that scheduling program task on CPU/GPU based heterogeneous systems is non-trivial.

Given a set of CPU and GPU devices in a target heterogeneous multicore, a scheduling strategy, the objective of the scheduling framework is to efficiently map all the tasks with the aid of the strategy to devices in target heterogeneous multicore such that the overall schedule makespan is minimized.

Here is a generic pseudocode depicting the sample usage of the framework. This algorithm 1 expects a set of `Kernel` objects denoted by $\mathcal{K}$ representing the task set. The framework maintains the counters $nCPU, nGPU$ as global variables.

Algorithm 1 Scheduling Framework

Input: $\mathcal{K}$ - A set of Kernel objects representing taskset, $nCPU, nGPU$

```
1: compute_requirement(K)
2: host_initialize(nCPU, nGPU)
3: input_data_initialize(K)
4: initialize_queues(K)
5: while queue set Q is not empty do
6:     if nCPU > 0 or nGPU > 0 then
7:         k, U ← select(Q, nCPU, nGPU),
8:         if nCPU > rCPU and nGPU > rGPU then
9:             mG ← ⌊nGPU / rGPU⌋
10:            nC ← ⌊mCPU / rCPU⌋
11:            dispatch_multiple(nC,mG,k,U)
12:        else
13:            dispatch(k,U)
```

Description of the variables and functions used in the pseudocode.

- $\mathcal{K}$: A set of $Kernel$ objects which represent the taskset
- $\mathcal{Q}$: Represents the $QueueSet$ object, which is a set of queues containing the $Kernel$
- $nCPU$: number of available CPU devices.
- $nGPU$: number of available GPU devices.
- $rCPU$: number of CPUs required to schedule all unprocessed tasks in parallel
- $rGPU$: number of GPUs required to schedule all unprocessed tasks in parallel
- $compute\_baseline\_requirements(\mathcal{K})$: This module takes as argument the set $\mathcal{K}$ of all Kernel objects and computes the initial values of $rCPU$ and $rGPU$. The computation of $rCPU$ and $rGPU$ is based on predicted static partition ratios by the Machine Learning frontend.

- *host_initialize*$(nCPU, nGPU)$: This function will create the requested number of OpenCL devices, builds two OpenCL contexts and creates an OpenCL Command Queue for each device.

- *input_data_initialize*$(\mathcal{K})$: This function takes as input a set of `Kernel` objects $\mathcal{K}$ and populates corresponding host arrays with valid data.

- *initialize_queues*$(\mathcal{K})$: This function initializes the set $\mathcal{Q}$ of task queues and populates each queue with member tasks as per a given task queue specification.

- *select*$(\mathcal{Q}, nCPU, nGPU)$: A user-defined function that selects a *kernel* from the *QueueSet* based on the schedule strategy which is to be dispatched. This returns a task t to be dispatched along with a mode $U \in \{cpu, gpu, mixed\}$ of dispatch.

- *dispatch* and *dispatch_multiple* : These functions have the same meaning as described in the API Specification.

The lines $1 - 4$ of the algorithm 1 initialize the objects required by the scheduling algorithm. After the initialization is done, we'll continuously try to dispatch the tasks from the $\mathcal{Q}$ as long as there is some task left in the $\mathcal{Q}$. The scheduler will dispatch a task only when there is at least one free device available in the system. The device availability information is provided by the counters $nCPU$ and $nGPU$ which are updated by $notify\_callback()$ based on device queue status. The selected task can be either dispatched to the bare minimum number of devices (one or two) while respecting its partition ratio or it can be dispatched to multiple devices in parallel by distributing its computation. This choice is decided by the values of $rCPU, rGPU, nCPU$ and $nGPU$. Scheduling a task to multiple devices (more than two) will reduce the number of available devices in the system and as a result will reduce the degree to which multiple tasks will be dispatched in parallel.The values

of nC and mG denote the maximum number of CPU and GPU devices which may be used per task so that the tasks in the taskset can be dispatched in parallel.

3.2.2 Scheduling Kernels with Dependencies

The algorithms and scheduling strategies discussed so far are applicable for task sets without dependencies. When there are dependencies between tasks they need to be handled differently. The dependent tasks can not be dispatched until their dependencies are satisfied. There will be a data transfer between dependent tasks and theeit dependencies.

Consider a set of OpenCL applications where each application is described in form of a task graph or a Directed Acyclic Graph (DAG) with each node representing a task and each directed edge representing task level dependencies. Fig. 3.3 depicts one such taskset consisting of two application DAGs. Once processed by the ML based partition prediction framework, each node in the DAG is labeled with modes of execution CPU (partition class 0), GPU (partition class 10), CPU/GPU (partition class 1-9). A method for partition aware scheduling of DAGs using the proposed framework is given as follows. The input for the method is a set $\mathcal{D}$ of DAGs. A set $\mathcal{F}$ (called frontier) comprising the start task of each DAG is initially constructed. The different phases of intialization like baseline requirement computation and setting up of queues for the tasks in $\mathcal{F}$ are then performed. The subsequent iterative steps for task dispatch at runtime are as follows.

- The tasks in $\mathcal{F}$ are independent by construction and can be scheduled using any of the strategies proposed earlier. Following, any of these strategies, the scheduler dispatches a task from the frontier $\mathcal{F}$ whenever there is any available device in the system as ascertained by the counters $nCPU$ and $nGPU$.
- When a task in the frontier finishes execution, it is removed from $\mathcal{F}$. In Fig. 3.3, the blue nodes denote tasks that have finished execution. The nodes

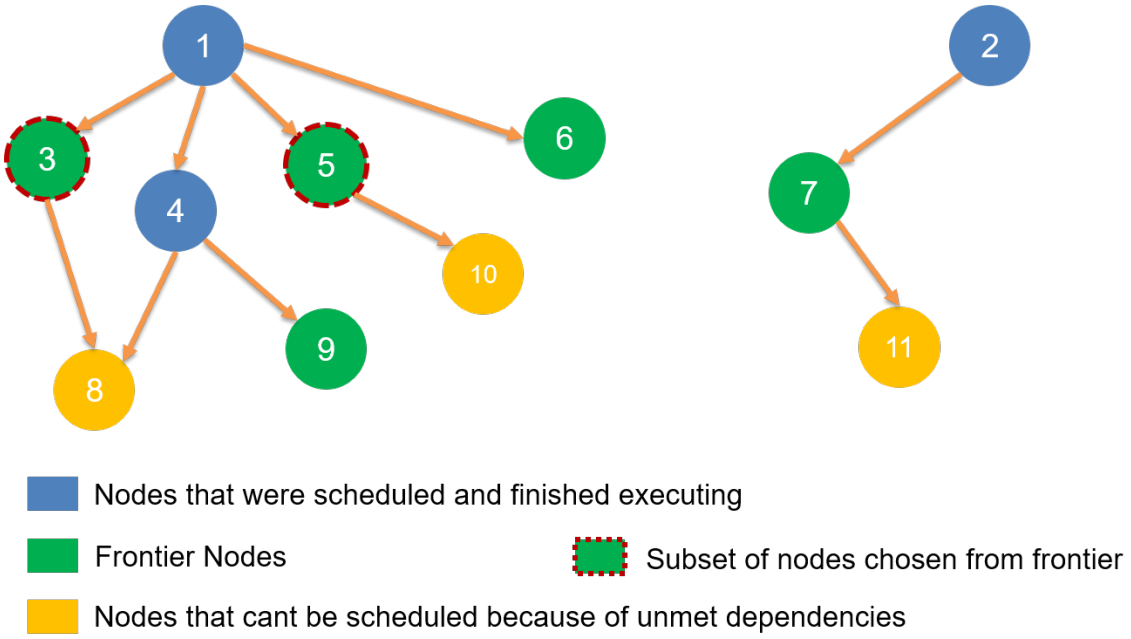

FIGURE 3.3: Kernels with dependencies.

in the frontier $\mathcal{F}$ are colored green. Once a task finishes execution in $\mathcal{F}$, it is colored blue and the $notify\_callback()$ function updates the frontier by adding successors of the completed tasks to $\mathcal{F}$.

- New tasks added to the frontier are enqueued to one of the queues in Q based on the scheduling heuristic used.The final dispatch decision of tasks to single or multiple devices is the same as that of the algorithm for scheduling independent tasks.

The pseudocode for the DAG scheduling method discussed above is given in Algorithm 2.

Algorithm 2 DAG Scheduling

Input: $\mathcal{D}$ - A set of DAGs representing task set $T, nCPU, nGPU$

1: $\mathcal{F} \leftarrow$ starting task of each DAG in $\mathcal{D}$
2: initialize($\mathcal{F}$)
3: compute_requirement($\mathcal{F}$)
4: $\mathcal{Q} \leftarrow$ initialize_queues($\mathcal{F}$)
5: **while** frontier $\mathcal{F}$ is not empty **do**
6: Update $nCPU, nGPU, rCPU, rGPU$
7: Update $\mathcal{F}$ with successors of tasks finished
8: Update queue set $\mathcal{Q}$ with new tasks in $\mathcal{F}$
9: **if** nCPU $>$ 0 and nGPU $>$ 0 **then**
10: k, U $\leftarrow$ select($\mathcal{Q}, nCPU, nGPU$),
11: **if** nCPU $>$ rCPU and nGPU $>$ rGPU **then**
12: $mG \leftarrow \lfloor \frac{nGPU}{rGPU} \rfloor$
13: $nC \leftarrow \lfloor \frac{mCPU}{rCPU} \rfloor$
14: dispatch_multiple(nC,mG,k,U)
15: **else**
16: dispatch(k,U)

Chapter 4

Scheduling Strategies

We have previously had an overview of how OpenCL Tasks are scheduled in Chapter 3. This chapter describes about the specific strategies that we have tested with the scheduling framework.

4.1 Strategies for Scheduling Independent Tasks

From pseudocode for a generic scheduling algorithm (Algorithm 1), it is evident that the way *select* function chooses the tasks will affect the overall span of the schedule. These strategies can make decisions using $nCPU, nGPU$, $rCPU, rGPU$ counters, the optimal partition value of the tasks expected using the machine learning frontend. The framework makes testing different scheduling strategies easy as the *select* function is easily pluggable. The user can define their own strategy in accordance with the parameters pertaining to a task and design their own scheduling algorithm by implementing the `select()` module. The user specifies $\mathcal{Q}$ and the available devices in the system $nCPU$ and $nGPU$, designs a set of rules for a particular scheduling strategy. Here we describe some of the proposed strategies.

4.1.1 Baseline Selection Algorithm

Algorithm 3 Baseline Selection

```
1: procedure SELECT(Q, nCPU, nGPU)
2:     if nCPU > 0 and nGPU > 0 then
3:         k ← Q.M.top()
4:         U ← partition
5:     else if nCPU > 0 then
6:         k ← Q.C.top()
7:         U ← cpu
8:     else
9:         k ← Q.G.top()
10:        U ← gpu
11:    return {k, U}
```

We first discuss the baseline algorithm shown in Algorithm 3. The task queue specification $\mathcal{Q}$ maintains three distinct priority queues $\mathcal{C}$, $\mathcal{G}$ and $\mathcal{M}$ which contains kernels that should execute only on the CPU, only on the GPU, and in a partitioned fashion between a CPU and GPU device as suggested by the ratio r_t respectively. Each of the three task queues are again sorted according to the feature value of total datasize transferred. This ensures that programs with smaller problem sizes are dispatched first, so that the overall waiting time for all the tasks in the task set is minimized. The main objective of this scheduling algorithm is to execute each and every kernel in such a way that its predicted partition ratio is respected. The selection of a task and its corresponding dispatch mode depends directly on the set of available devices in the system. Following are the set of rules for this strategy.

- If both types of devices are available in the system, a task from queue $\mathcal{M}$ is selected. and the dispatch mode U is set to *partition* (lines $3-4$).
- For cases when devices of only one particular type are present in the system, tasks from queues C and G are selected and corresponding dispatch modes U are set to respective devices (lines $5-10$).

4.1.2 Lookahead Selection Algorithm

Algorithm 4 Lookahead Selection

```
1: procedure SELECT(Q, nCPU, nGPU)
2:     if pending_task is non-empty then return pending_task
3:     if nCPU > 0 and nGPU > 0 then
4:         k1 ← Q.M.pop()
5:         k2 ← Q.M.pop()
6:         if k1.r >= 8 and k2.r <= 2 then
7:             pending_task ← {k2, cpu}
8:             k ← k1
9:             U ← gpu
10:        else if k1.r <= 2 and k2.r >= 8 then
11:            pending_task ← {k1, cpu}
12:            k ← k2
13:            U ← gpu
14:        else if k1.ECO < k2.ECO then
15:            k ← k1
16:            U ← partition
17:            Q.M.push(k2)
18:        else
19:            k ← k2
20:            U ← partition
21:            Q.M.push(k1)
22:    else if nCPU > 0 then
23:        k ← Q.C.top()
24:        U ← cpu
25:    else
26:        k ← Q.G.top()
27:        U ← gpu
28:    return {k, U}
```

The lookahead algorithm (Algorithm 4) is slightly sophisticated in the sense, that workloads which were to be executed in a partitioned fashion between CPU and GPU devices, may be executed on devices of a particular type instead. The heuristic helps in selecting the task as well as determining its dispatch mode. The task queue specification $\mathcal{Q}$ is the same as that of the Baseline algorithm maintaining three

queues $\mathcal{C}$, $\mathcal{G}$ and $\mathcal{M}$ here as well. The measure s_t is now slightly more sophisticated than before. The total data size transferred is restrictive and does not capture the computational intensiveness of a task completely. We introduce the *ECO* measure in this context.

Definition 4.1. The **Estimated Computation Overhead** of an OpenCL task is a static measure of the total number of floating point and fixed point operations in the task computed using the following equation.

$$ECO = num_{work-items} * (I + F + \sum_{l \in L} n_l * (I_l + F_l)) \tag{4.1}$$

where $num_{work-items}$ represents the total number of work-items launched, L is the set of loops present in the OpenCL kernel, I, F are the total number of fixed point and floating point operations which are not part of any loop body, n_l is the loop bound of a loop $l \in L$, I_l and F_l are the number of fixed point and floating point operations for a loop $l \in L$.

The total number of work-items is actually the problem size and determines the total volume of fixed and floating point computation for a particular task. As a rule of thumb, tasks with high ECO values should be partitioned, whereas the ones with low ECO values should be dispatched to single device. The set of rules for the Lookahead scheduling strategy is listed as follows.

- Whenever devices of both types are available in the system, the top two elements of the queue $\mathcal{M}$ are chosen and compared for final selection (lines $4-21$). If it is observed that the two tasks are highly biased towards a particular device (one has partition class value greater than 8 and the other has a partition class value less than 2), the dispatch modes U are set to their respective single devices (lines $6-13$). One of the tasks and its corresponding dispatch mode is returned directly. The other task is stored in a variable called *pending_task*.

The corresponding task will be returned immediately for dispatching the next time the *select*() module is called(line 1).

- If a task has not been selected by the above rule, the task t which has a lower ECO value is selected and its corresponding dispatch mode is set to *partition* (lines $14-21$). The task not selected is pushed back into the queue.
- For cases when devices of only one particular type the rules are the same as that of the baseline algorithm (lines $23-27$).

4.1.3 Adaptive Bias Selection Algorithm

In the Adaptive Bias algorithm (Algorithm 5), the task queue specification $\mathcal{Q}$ maintains four queues $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{G}$ and $\mathcal{C}$. Applications whose partition class values are in the range $(1-4)$ are enqueued in $\mathcal{M}_1$ while tasks whose partition class values are in the range $(5-9)$ are enqueued in $\mathcal{M}_2$. These represent the tasks which are biased towards a CPU and biased towards a GPU respectively. The maximum ECO value of the kernels in the task set is also maintained as ECO_{Max}. This strategy is also along similar lines of the lookahead algorithm where the dispatch mode for the task is determined by the heuristic. But the difference here lies in explicit specification of the task queues in terms of how biased a task is towards a device. The set of rules for this particular strategy is listed as follows.

- When both CPU and GPU devices are available in the system, tasks from queues $\mathcal{M}_1$ and $\mathcal{M}_2$ are chosen for selection (lines $4-31$). If 80% of the computation or greater is mapped to a particular device for both the tasks $k1$ and $k2$ the dispatch modes for both the tasks will be changed from *partition* to the device type the task is biased towards (lines $7-10$). Here again, one kernel is returned by the select function whereas the other kernel is stored in *pending_task* which will be selected immediately the next time the *select* module is called.

- For cases where computation of the tasks $k1$ and $k2$ mapped to a device is greater than or equal to 60 % of the total computation, a more sophisticated decision is made while dispatching tasks to devices (lines $11-22$). The dispatch mode for execution of a task will now depend on the computational intensity of the task relative to the other tasks in the task set. We utilize the *ECO* measure of the kernels here during the selection process. If the ECO measure of a task is within a threshold factor t of the maximum ECO value ECO_{Max} of the task set, the dispatch mode of a task is changed from *partition* to the device type the task is biased towards.

- If no task has been dispatched till now by the above two rules, the task with the lower ECO value should be selected and the corresponding dispatch mode is set to *partition* (lines $24 - 31$).

- For cases when devices of only one particular type are present in the system, the rules are the same as that of the previous two algorithms (lines $33 - 37$).

Algorithm 5 Adaptive Bias Selection

```
1: procedure SELECT(Q, nCPU, nGPU)
2:     if pending_task is non-empty then return pending_task
3:     if nCPU > 0 and nGPU > 0 then
4:         is_dispatched == 0
5:         k1 ← Q.M1.pop()
6:         k2 ← Q.M2.pop()
7:         if k1.r <= 2 and k2.r >= 8 then
8:             pending_task={k1, cpu}
9:             k ← k2
10:            U ← gpu
11:        else if k1.r <= 4 and k2.r >=6 then
12:            if k1.ECO < t*ECO_Max then
13:                is_dispatched = 1
14:                k ← k1
15:                U ← cpu
16:            if k2.ECO < t*ECO_Max then
17:                is_dispatched = 1
18:                if k is non-empty then
19:                    pending_task ← {k2, gpu}
20:                else
21:                    k ← k2
22:                    U ← gpu
23:        if is_dispatched ≠ 1 then
24:            if k1.ECO < k2.ECO then
25:                k ← k1
26:                U ← partition
27:                Q.M2.push(k2)
28:            else
29:                k ← k2
30:                U ← partition
31:                Q.M1.push(k1)
32:    else if nCPU > 0 then
33:        k ← Q.C.top()
34:        U ← cpu
35:    else
36:        k ← Q.G.top()
37:        U ← gpu
38:    return {k, U}
```

4.2 Strategies for Scheduling Tasks with Dependencies

4.2.1 Contraction Algorithm

Algorithm 2 simply uses existing strategies with additional bookkeeping for handling dependencies. However, smarter strategies can be easily conceived in the context of partition aware DAG scheduling. For example, it makes sense to map a subset of dependent tasks which are highly GPU biased to a single GPU if there exists a considerable amount of data transfer. This is because the data processed by a task is kept in the GPU for all the successor tasks to process and update thereby removing the overhead in transferring data back and forth to the CPU. In a similar way, a subset of tasks which are highly CPU biased maybe mapped to a single CPU. Efficient heuristics must be designed to ascertain whether mapping subsets of tasks to single devices would prove to be feasible. This can be determined statically by performing a dependency analysis of the tasks in a DAG annotated with execution time estimates. We provide a graph transformation heuristic which performs such merging of nodes in the DAGs before the partition aware scheduling is actually performed.

Algorithm 6 Task Contraction Algorithm

procedure H($\mathcal{D}$, $feat$, d, p)
  $H \leftarrow$ Construct subgraph of D with nodes at depth d and d+1
  $C \leftarrow$ Connected components of H
  for each conneted component $C_i \in C$ **do**
    $R \leftarrow$ tasks in C_i which were at depth d in D
    $S \leftarrow$ successors of tasks in R
    for each task t in R and S **do**
      $f_t \leftarrow max(\frac{p_t * feat.ECO(t)}{10 * FLOP_{GPU}}, \frac{(10-p_t) * feat.ECO(t)}{10 * FLOP_{CPU}})$
    for each task r in R **do**
      for each task s in Succ(r) **do**
        $dt(r, s) \leftarrow \frac{DT(r,s)}{BW}$
    $DT^{CPU}_{diff} \leftarrow \sum_{r \in R} \sum_{s \in Succ(r)} dt(r, s) * (\frac{p_s}{5})$
    $DT^{GPU}_{diff} \leftarrow \sum_{r \in R} \sum_{s \in Succ(r)} dt(r, s) * (\frac{p_s}{5} - 1)$
    $f_{max} \leftarrow \max_{t \in S}((\max_{t' \in ancestors(t)} f_{t'}) + f_t)$
    $EX^{CPU} \leftarrow \sum_{r \in R} \frac{feat.ECO(r)}{FLOP_{CPU}} + \sum_{s \in S} \frac{feat.ECO(s)}{FLOP_{CPU}}$
    $EX^{GPU} \leftarrow \sum_{r \in R} \frac{feat.ECO(r)}{FLOP_{GPU}} + \sum_{s \in S} \frac{feat.ECO(s)}{FLOP_{GPU}}$
    $EX^{CPU}_{diff} \leftarrow EX^{CPU} - f_{max}$
    $EX^{GPU}_{diff} \leftarrow EX^{GPU} - f_{max}$
    if $DT^{CPU}_{diff} > EX^{CPU}_{diff} and DT^{GPU}_{diff} > EX^{GPU}_{diff}$ **then**
      $DT \leftarrow \sum_{r \in R} \sum_{s \in Succ(r)} dt(r, s)$
      if $DT + EX^{GPU} > EX^{CPU}$ **then**
        Contract(C_i, CPU)
      else
        Contract(C_i, GPU)
    else if $DT^{CPU}_{diff} > EX^{CPU}_{diff}$ **then**
      Contract(C_i, CPU)
    else if $DT^{GPU}_{diff} > EX^{GPU}_{diff}$ **then**
      Contract(C_i, GPU)

Algorithm 6 illustrates the functioning of a heuristic function $\mathcal{H}$ which decides whether nodes belonging to a DAG $\mathcal{D}$ at a depth d and its successors at depth $d + 1$ should be merged or not. A merge operation implies mapping a subset of tasks to a single GPU device or a single CPU device. The heuristic function $\mathcal{H}$ takes the DAG D, the depth d of tasks to be considered, the set of feature values of all the tasks in the DAG given by $feat$ and the partition class values p_t for each task

$t \in D$. The algorithm first constructs a subgraph H which considers only the nodes at depth d and $d+1$ (line 2). The subgraph H does not have edges between nodes at depth d and its predecessors and nodes at depth $d+1$ and its successors. Once the subgraph H is constructed, the connected components for H are obtained (line 3). The decision for mapping each component C_i to a single device is taken using execution time estimates and data transfer time estimates between the tasks in each connected component.

The tasks in C_i which were at depth d in D is stored in R (line 5). The tasks which are successors of tasks in R at depth $d+1$ are stored in S. (line 6). The execution time estimate for a task t, given by f_t, is estimated using the ECO value of the task $feat.ECO(t)$, the FLOPs specification for a device ($FLOP_{CPU}$ and $FLOP_{GPU}$) and the respective partition class value for the task p_t (lines 7-8). For each task t, $(\frac{p_t}{10}) * feat.ECO(t)$ number of operations are executed on the GPU and $(\frac{10-p_t}{10}) * feat.ECO(t)$ number of operations are executed on the CPU in parallel. The execution time estimates on each device is obtained by dividing the number of operations with the respective FLOPs specification of the device. The maximum of the two execution time estimates represents the overall execution time estimate f_t for the task t.

The estimate for data transfer overhead between pairs of tasks $(r,s)\, r \in R, s \in Succ(r)$ is ascertained using the total data size transferred $DT(r,s)$ and the bandwidth specification BW (lines 9-11). The decision for contracting a connected component is now taken based on the difference in total time between the scenarios as discussed below.

- **Scenario 1**: Tasks are executed in parallel with their original partition class values and data is transferred back and forth to the CPU for each task pair (r,s) where s at depth $d+1$ is a successor of task r at depth d.
- **Scenario 2**: The set of tasks in a connected component are combined i.e. tasks are executed sequentially in one single CPU device.

- **Scenario 3**: The set of tasks in a connected component are combined i.e. tasks are executed sequentially in one single GPU device.

For each connected component C_i, we compute the timing difference between scenarios 1 and 2/3 in terms of data transfer overhead and execution time of the tasks. The quantity DT_{diff}^{CPU} (line 12) represents the difference in data transfer overhead between scenarios 1 and 2. Similarly, the quantity DT_{diff}^{GPU} (line 13) represents the difference in data transfer overhead between scenarios 1 and 3. For any connected component C_i, the execution time differences between the scenarios 1 and 2/3 are computed as follows.

For scenario 1, the total execution time f_{max} is calculated (line 14) by assuming that tasks in R and S have sufficient resources to execute in parallel. The total execution time for the tasks in the two levels d and $d+1$ is computed as $f_{max} = \max_{t \in S}((\max_{t' \in ancestors(t)} f_{t'}) + f_t)$ where f_t is the ECO based execution time estimate of some task $t \in S$ and $f_{t'}$ is the ECO based execution time estimate of some task t' which is one of the ancestors of t, i.e. $t' \in ancestors(t)$. Thus, f_{max} provides the (relative) completion time for component C_i by taking into account the execution time estimates of tasks $\in C_i$ at depth $d+1$ and its ancestors at depth d. For scenarios 2 and 3, the total execution times EX^{CPU} and EX^{GPU} are computed respectively as follows.

- EX^{CPU} : The quantity is computed by assuming that all tasks in R and S are mapped to a single CPU. Therefore, the execution time estimate for each task t now becomes $\frac{feat.ECO(t)}{FLOP_{CPU}}$. The total sum of these execution time estimates of tasks in R and S is computed and stored in EX^{CPU} (line 15).

- EX^{GPU} : The quantity is computed by assuming that all tasks in R and S are mapped to a single GPU. Therefore, the execution time estimate for each task t now becomes $\frac{feat.ECO(t)}{FLOP_{GPU}}$. The total sum of these execution time estimates of tasks in R and S is computed and stored in EX^{GPU} (line 16).

The difference between execution times of scenario 1 and scenario 2 is represented by the quantity EX_{diff}^{CPU} (line 17). The difference between execution times of scenario 1 and scenario 3 is represented by the quantity EX_{diff}^{GPU} (line 18).

The Contract function is called which decides whether all the tasks in C_i should be combined and assigned a single CPU or GPU device depending on the values of the quantities $DT_{diff}^{CPU}, DT_{diff}^{GPU}$ and $EX_{diff}^{CPU}, EX_{diff}^{GPU}$ (lines 19-28). The conditions for contracting a connected component C_i are elaborated as follows.

- If it is observed that DT_{diff}^{CPU} is more than EX_{diff}^{CPU} and DT_{diff}^{GPU} is more than EX_{diff}^{GPU} simultaneously, the implication is that scenario 1 takes more time than 2 as well as 3. In that case, the decision of contracting C_i to a CPU or the GPU is made as follows. The quantity DT is computed which represents the total data transfer time involved in executing tasks in Scenario 3. The total execution time on the GPU is represented by $DT + EX^{GPU}$. If this quantity is more EX^{CPU} , C_i is contracted and mapped to the CPU. Otherwise, it is contracted and mapped to the GPU.

- If it is observed that only DT_{diff}^{CPU} is more than EX_{diff}^{GPU} , C_i is contracted and mapped to the CPU.

- If it is observed that only DT_{diff}^{GPU} is more than EX_{diff}^{GPU} , C_i is contracted and mapped to the GPU.

- If none of the above apply, the connected component C_i is left unchanged.

The heuristic $\mathcal{H}$ is applied to nodes of each DAG $\mathcal{D}$ for every depth in the graph and is transformed by replacing subsets of nodes with one single node. The transformed graph can easily be passed as input to the algorithm which we discussed earlier. We note that the heuristic described here implements a lookahead of depth of two in the DAG. A recursive formulation of this can easily generalise to any n level lookahead.

Chapter 5

PySchedCL Framework

5.1 Introduction

The OpenCL Runtime API is an elaborate low-level API. Any C/C++ OpenCL host program requires many API calls to execute even a very simple kernel. The presented generic scheduling framework abstracts away all the trivial stuff and provides a simple and elegant way to run a kernel, schedule set of independent kernels or schedule a set of dependent kernels. With this framework if the user describes the specification of each kernel (everything the framework needs to know inorder to run the kernel) as a JSON file it can dispatch the kernel to an available OpenCL device.

5.2 Getting Started

Following is the specification of VectorAdd kernel which takes two vectors and adds them.

```
1 {
2   "name": "VectorAdd",
3   "src": "VectorAdd.cl",
4   "workDimension": 1,
```

```
5   "globalWorkSize": "[dataset]",
6   "inputBuffers": [
7 {"type": "float", "size": "dataset", "break": 1, "pos": 0},
8       {"type": "float", "size": "dataset", "break": 1, "pos": 1}
9 ],
10  "outputBuffers": [
11 {"type": "float", "size": "dataset", "break": 1, "pos": 2}
12 ],
13  "varArguments": [
14 {
15        "type": "float",
16        "value": "partition_round(dataset, size_percent)",
17        "pos": 3
18      }
19 ],
20  "partition": 10,
21 }
```

Listing 5.1: Generated Kernel

The above specification describes the following kernel

```
1
2 __kernel void VectorAdd(__global const float* a,
3     __global const float* b,
4     __global float* c, int iNumElements)
5 {
6     // get index into global data array
7     int iGID = get_global_id(0);
8
9     // bound check (equivalent to the limit on
10    //a 'for' loop for standard/serial C code
11    if (iGID >= iNumElements)
12    {
13        return;
14    }
15
16    // add the vector elements
17    c[iGID] = a[iGID] + b[iGID];
18 }
```

Listing 5.2: Generated Kernel

It can be observed that the specification tries to describe

- the name of the kernel
- path to the corresponding OpenCL kernel file.
- how the kernel expects its arguments to be.
- how the kernel arguments depend on the size of data.
- globalworksize.
- optimal way to partition it among CPUs and GPUs.

With the help of specification the following kernel can be dispatched easily by the following host code.

```
1 import numpy as np
2 import pyschedcl as fw
3
4 cmd_qs, ctxs, gpus, cpus = fw.host_initialize(4, 2)
5 k = fw.Kernel(info)
6 k.build_kernel(gpus, cpus, ctxs)
7 k.random_data()
8 k.dispatch(0, 0, ctxs, cmd_qs)
```

Listing 5.3: Generated Kernel

5.3 Overview of Framework

The framework was developed in Python. Python was chosen because of its portability over different platforms and its dynamic capabilities. PySchedCL relies heavily on PyOpenCL[3] which is a wrapper over the OpenCL C API described in Chapter 2. It uses NumPy[4] for handling input and output datasets of kernels, NetworkX[1] to maintain dependencies among various tasks as a Directed Acyclic Graph, Plotly[2] for visualizing results.

The figure 5.1 depicts an general overview of scheduling tasks with the framework. It takes a set of specification files as input. Once it parses the specification files

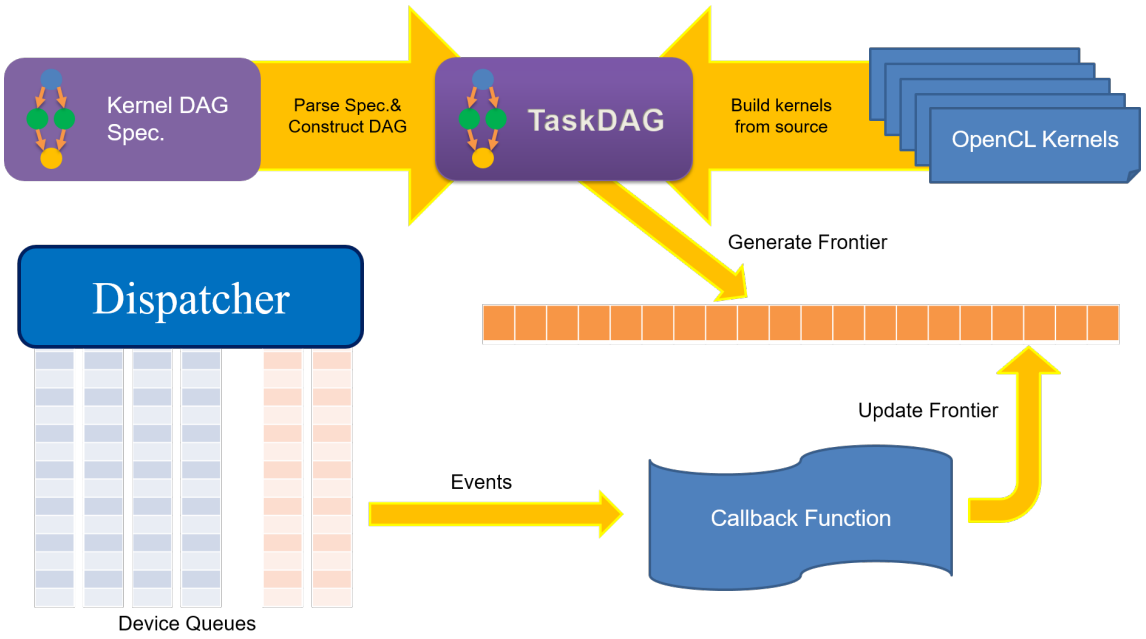

FIGURE 5.1: Scheduling Workflow with Framework

it obtains corresponding *Kernel* source files. The complete specification of these Kernel Specification Files is described in Chapter 6. Then the framework initializes various OpenCL constructs required for dispatching kernels. We maintain the dependencies among tasks (if any) by constucting a *DAG* data structure. From this *DAG* structure a frontier of tasks that have no unmet dependencies is generated. If the tasks has no dependencies then frontier will be the entire set of tasks. From these tasks we select a subset of them to dispatch based on number of available OpenCL Devices. Various strategies for selecting these tasks are highlighted in Chapter 5. Once these tasks are selected we need to bootstrap them before they can be dispatched. This bootstrap phase includes activities like compiling and building programs against OpenCL devices, loading/generating input data, creating buffers for data transfer etc. Then these tasks are handed to dispatcher which dispatches them to corresponding device queues. Once these tasks have finished execution then devices trigger a callback function, which profiles the operations, notifies the framework that these tasks have completed. Now any tasks that have these tasks as dependencies enter the frontier. This flow continues until all tasks are finished.

The implementation details of the framework are documented in the next chapter.

Chapter 6

Design and Implementation Details

6.1 Kernel Specification File

The Kernel Specification file for the individual kernels is a JSON file with the attributes as described in 6.1

The Kernel Specification file for a set of kernels with dependencies will be a JSON file containing list of Kernels Specification for independent kernels as descibed in Table 6.1.

Attribute	**Type**	**Description of the value**
name	String	Name of OpenCL Kernel.
src	String	Path to the source of the OpenCL Kernel file.
workDimension	Integer	Number of dimensions of the work-dimension of the kernel.

partition	Integer	An integer in range [0..10] which describes how to partition the kernel between CPU and GPU. This value will be generated by ML based partition prediction framework.
globalWorkSize	String	A python list expression denoting the Global Work Size Dimensions of the Kernel.
inputBuffers	List	A list of dictionaries containing information of read-only buffers with the structure described in Table 6.2. These will be the arguments with `__global` modifier and whose values are not modified in the kernel.
ioBuffers	List	A list of dictionaries containing information of read-write buffers with the structure described in Table 6.2. These will be the arguments with `__global` modifier and whose values are modified in the kernel.
outputBuffers	List	A list of dictionaries containing information of write-only buffers with the structure described in Table 6.2. These will be the arguments with `__global` modifier and whose values are initialized in the kernel.
varArguments	List	A list of dictionaries containing information regarding variable arguments passed to the kernel with the structure described in Table 6.3.
localArguments	List	A list of dictionaries containing information regarding local arguments passed to the kernel with the structure described in Table 6.5. These will be the arguments with `__local` modifier.
eco	Dictionary	A dictionary mapping between size of dataset and Estimated Computation Overhead values. These will be generated by ML based partition prediction framework.

id	String	This is the string which should be able to uniquely identify the kernel among the set of kernels. This is an optional field.
depends	List	A list of strings which must be *id*s of other kernels. This represents dependencies between kernels. This field is only used when specifiying set of kernels with dependencies.

Table 6.1: Description of Kernel Specification File

Attribute	**Type**	**Description of the value**
pos	Integer	Zero-based index of this Kernel argument.
size	(Integer | String)	Number of elements in this array.
type	String	A string representing OpenCL Datatype of the argument.
break	Integer(0|1)	Whether this array should be broken during partitioning.
from	Dictionary	A dictionary containing fields `kernel` and `pos` whose values will be the *id* of the kernel (String) this data is obtained from and position of the argument in that kernel (Integer) respectively. This field is only used when specifiying set of kernels with dependencies.

Table 6.2: Buffer Information Specification

Attribute	**Type**	**Description of the value**
pos	Integer	Zero-based index of this Kernel argument.
type	String	A string representing OpenCL Datatype of the argument.
value	String	A string containing python expression which when evaluated against `dataset` and `size_percent` gives the value of the argument at that partition.

Table 6.3: Specification of Variable Arguments

Attribute	**Type**	**Description of the value**
pos	Integer	Zero-based index of this Kernel argument.
type	String	A string representing OpenCL Datatype of the argument.
size	(Integer | String)	Number of elements in this array.

Table 6.4: Specification of Local Arguments

Attribute	**Type**	**Description of the value**
pos	Integer	Zero-based index of this Kernel argument.
type	String	A string representing OpenCL Datatype of the argument.
size	(Integer | String)	Number of elements in this array.

Table 6.5: Specification of Local Arguments

6.2 PySchedCL Framework

6.2.1 pyschedcl.Kernel Class

This Python class encapsulates all data related to an OpenCL kernel task. The following methods are used to dispatch an Opencl kernel task represented by `Kernel` class.

6.2.1.1 Attributes

dataset	An integer representing size of the data on which kernel will be dispatched.
id	An id that is used identify a kernel uniquely.
eco	A dictionary mapping between size of dataset and Estimated Computation Overhead
name	name of the Kernel
src	Path to the Kernel source file.
partition	An integer denoting the partition class of the kernel.
work_dimension	Work Dimension of the Kernel.
global_work_size	A list denoting global work dimensions along different axes.
local_work_size	A list denoting local work dimensions along different axes.
buffer_info	Properties of Buffers
input_buffers output_buffers io_buffers	Dictionaries containing actual $cl.Buffer$ objects.

data	Numpy Arrays maintaining the input and output data of the kernels.
buffer_deps	Dictionary mapping containing buffer dependencies.
variable_args	Data corresponding to Variable arguments of the kernel.
local_args	Information regarding Local Arguments of the kernel.
kernel_objects	Dictionary mapping between devices and compiled and built *pyopencl.Kernel* objects.
events	Dictionary containing *pyschedcl.KEvents*.
source	String containg contents of kernel file.
clevents	Dictionary containing *pyopencl.Event*s.

Table 6.6: Member Variables of Kernel Class

6.2.1.2 Methods

- `get_num_global_work_items(self)` : Returns the total number of global work items based on global work size.
- `eval_vargs(self, partition, size_percent, offset_percent, reverse, exact, total)` : Evaluates variable kernel arguments from the specification file if they are an expression against *size_percent*.
- `get_partition_multiples(self)` : This method returns a list of numbers based on global work size and local work size according to wich the partiton sizes will be determined.
- `build_kernel(self, gpus, cpus, ctxs)` : Creates programs from source and builds them for each device. *gpus*, *cpus* are to lists of devices

- `random_data(self)` : Populates all host input arrays with random data.
- `load_data(self, data)` : Populates all host input arrays with given data.
- `get_data(self, pos)` : Returns the data of a particular kernel argument given its parameter position in the kernel.
- `get_buffer_info_location(self, pos)` : Returns location of $buffer\_info$ given its parameter position in the kernel. This method is used in making reusable buffers.
- `get_buffer_info(self, pos)` : Returns $buffer\_info$ given its parameter position in the kernel.
- `get_buffer(self, pos)` : Returns $pyopencl.Buffer$ objects given its parameter position in the kernel.
- `get_slice_values(self, buffer_info, size_percent, offset_percent, **kwargs)` : Returns Element offset, size for a buffer based on $size\_percent$ and $offset\_percent$.
- `create_buffers(self)` : Creates Input and Output Buffers for each Context for all required arguments.
- `set_kernel_args(self)` : Sets the argument values for the corresponding arguments of the kernel.
- `enqueue_write_buffers(self, queue, q_id, obj, size_percent, offset_percent, deps)` : Enqueue commands to write from given $size\_percent$ of host memory at given $offset\_percent$ to buffer objects.
- `enqueue_nd_range_kernel(self, queue, q_id, obj, size_percent, offset_percent)` : Enqueues the command to execute a kernel on a device.
- `enqueue_read_buffers(self, queue, q_id, obj, size_percent, offset_percent)` : Enqueue commands to read from a buffer object and write to given $size\_percent$ of host memory at given $offset\_percent$.

- `dispatch(self, gpu, cpu, ctxs, cmd_qs, dep=None, partition=None, callback=bla` : This method is used to dispatch a kernel task to a single GPU/CPU or in a partitioned way across GPU-CPU as specified by *partition* argument or partition it using the value generated by the ML frontend. A callback function can also be specified which will run once the kernel is finished.

- `dispatch_multiple(self, gpus, cpus, ctxs, cmd_qs, dep=None, partition=None, callback=blank_fn)` : This method is used to dispatch a kernel task across multiple GPUs and CPUs in a partitioned way across GPU-CPU, distributing the task as specified by *partition* argument or partition it using the value generated by the ML frontend.

- `get_device_requirement(self)` : Returns total nuber of devices required for dispatching the kernel according to its partition value.

6.2.2 pyschedcl.KEvents Class

Maintains different timestamps regarding Kernel execution.

load_start	Timestamp at which loading data from dependent kernels and buffers started. (Host).
dispatch_start	Start time of the dispatch (Host).
dispatch_end	End time of the dispatch (Host).
write_submit	Submit time of the $enqueue\_write\_buffers$ (Device).
write_start	Start time of the $enqueue\_write\_buffers$ (Device).
write_end	Finish time of the $enqueue\_write\_buffers$ (Device).
ndrange_start	Start time of the $enqueue\_nd\_range$ (Device).
ndrange_end	Finish time of the $enqueue\_nd\_range$ (Device).
read_start	Start time of the $enqueue\_read\_buffers$ (Device).

read_end	Finish time of the $enqueue\_read\_buffers$ (Device).
kernel_name	Name of the kernel.
kernel_id	Unique kernel id.
dispatch_id	Unique dispatch id.

TABLE 6.7: Member Variables of KEvents Class

6.2.3 pyschedcl.Task Class

This class is mainly used while scheduling dependent kernels. A $pyschedcl.Task$ can contain one or more $pyschedcl.Kernel$s. All Kernels belonging to a task will be dispatched to the same device.

6.2.3.1 Attributes

kernels	A set of $pyschedcl.Kernel$s that are part of this task.
dev_requirement	Total device requirement of all kernels in this task
finished_kernels	List of finished kernels in the task.
free_kernels	List of kernels that don't have any unmet dependencies.
recently_added_kernels	List of kernels whose dependencies have just been satisfied.
partition	Partition value of the kernels in this task.

TABLE 6.8: Member Variables of Task Class

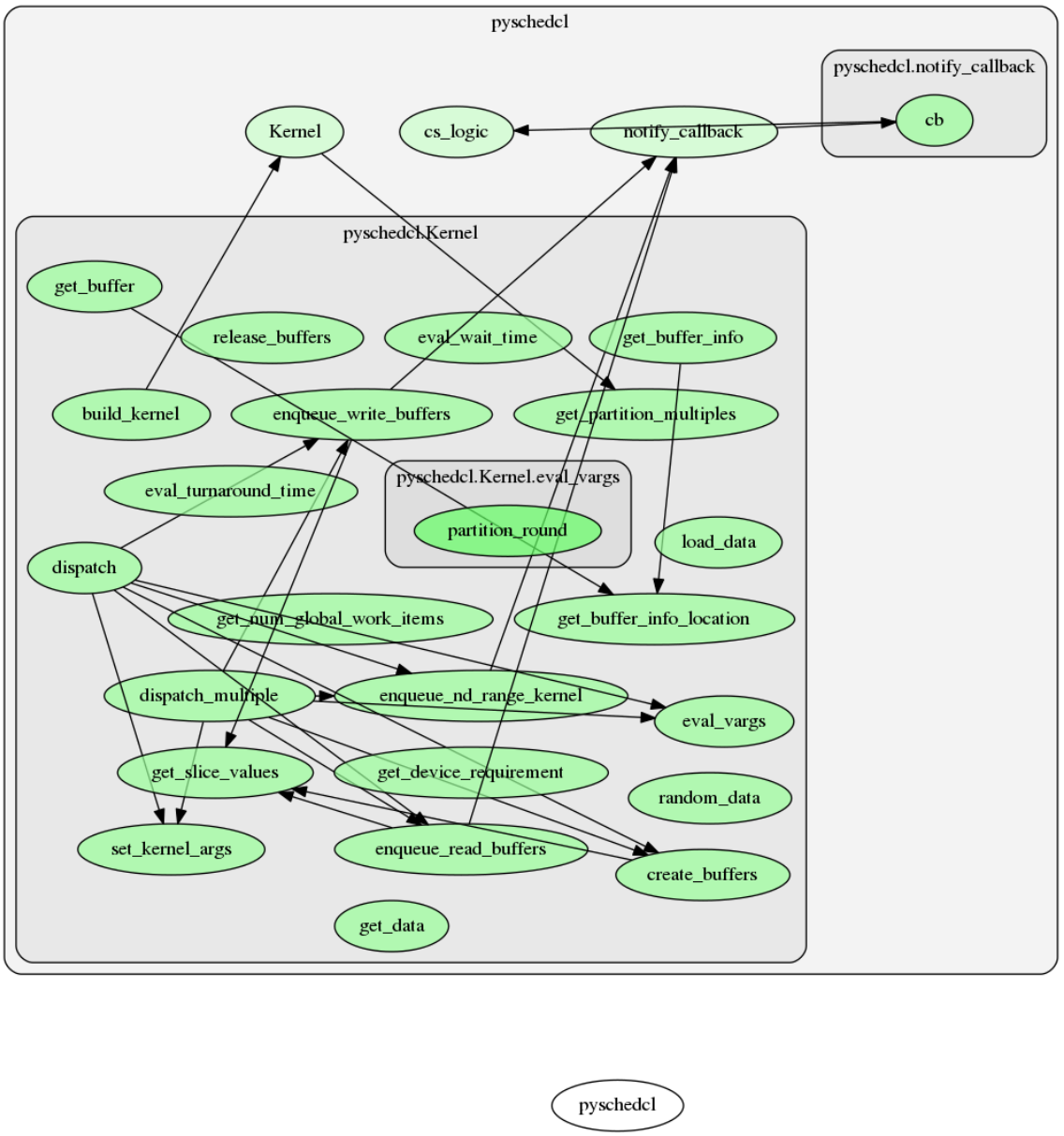

FIGURE 6.1: pyschedcl.Kernel Call Graph

6.2.3.2 Methods

- `load_dependent_data_and_buffers(self, dag, kernel_id)` : Given a *pyschedcl.Kernel* id and a *pyschedcl.TaskDAG* it determines the dependencies beween the given kernel and other kernels from all tasks. It loads dependent data from other tasks and reuses the dependent buffers from kernels within the same task.

- `prepare_kernel(self, kid, dag)` : Given Kernel id it prepares Kernel for dispatch by modifying properties of its buffers, so that buffers can be reused wherever possible.
- `build_kernels(self, gpus, cpus, ctxs)` : Builds all kernels in the task.
- `dispatch_single(self, dag, gpu, cpu, ctxs, cmd_qs, callback=blank_fn, *args, **kwargs)` : Dispatches one kernels from this task to a given single device. Assumes appropriate independent data is already loaded.
- `remove_kernel(self, kernel)` : Removes the give kernel from this task.
- `add_kernels_from_task(self, task)` : Adds all kernels from a given task to itself.
- `modify_device_requirement(self, kernels, op=operator.iadd)` : Modifies the divice requirement of the task based on the given kernels. This method will be called whenever some kernels are either added or removed from the task.
- `get_device_requirement(self)` : Returns the device requirement of this task.
- `get_first_kernel(self)` : Returns a kernel from this task.
- `get_kernels(self)` : Returns the set of kernels contained in this task.
- `get_kernel_ids(self)` : Returns the kernel ids of the kernels contained in this task.
- `get_kernels_sorted(self, dag)` : Given the *pyschedcl.TaskDAG* this method topologically sorts the kernels contained in this task and returns them as a list.
- `get_kernel_ids_sorted(self, dag)` : Given the *pyschedcl.TaskDAG* this method topologically sorts the kernels contained in this task and returns their kernel ids as a list.

- `is_supertask(self)` : A task is considered a supertask when it has more than one kernel. This method checks whether there are more than one kernel in the task.
- `is_finished(self)` : Checks whether all the kernels in this task have finished execution.
- `update_finished_kernels(self, kernel, dag, *args, **kwargs)` : This method will be called whenever a kernel belonging to this task finishes execution. When called this method updates $finished\_kernels$ and $free\_kernels$ accordingly so that tasks dependent on this kernel can then be dispatched.
- `refresh_free_kernels(self, dag)` : Checks $finished\_kernels$ and updates $free\_kernels$
- `get_some_free_kernel(self)` : Removes and returns a free kernel from $free\_kernels$
- `get_kernel_parents(self, kernel_id, dag)` : Given TaskDAG and a kernel id this method returns list of all the kernels that this kernel depends upon.
- `get_kernel_children(self, kernel_id, dag)` : Given TaskDAG and a kernel id this method returns list of all the kernels that are dependent upon this kernel.

6.2.4 pyschedcl.TaskDAG Class

This class maintains the actual dependencies between kernels and tasks and provides API to modify the tasks and their dependencies.

6.2.4.1 Attributes

kernels	Dictionary mapping between kernel ids and kernels.

tasks	Dictionary mapping between kernel ids and *pyschedcl.Task*s
finished_kernels	Set of finished kernels.
finished_tasks	Set of finished tasks.
free_kernels	A list of kernels that don't have any unmet dependencies.
free_tasks	A list of tasks that don't have any unmet dependencies.
processing_tasks	List of tasks that are currently being processed.
skeleton	A *netwokx.DiGraph* representing kernels and their dependencies.
G	A *netwokx.DiGraph* representing tasks and their dependencies.
recently_added_kernels	A list of kernels whose dependencies have just been satisfied.
recently_added_tasks	A list of tasks whose dependencies have just been satisfied.

TABLE 6.9: Member Variables of TaskDAG Class

6.2.4.2 Methods

- `update_dependencies(self, task)` : This method updates the task dependencies. This will be called whenever a task is modified. It adds or remove edges to task dag based on skeleton kernel dag for the given task.
- `get_skeleton_subgraph(self, kernel_ids)` : Returns a *networkx.DiGraph*
- `update_finished_kernels(self, kernel_id, *args, **kwargs)` : This method must be called whenever a kernel finishes execution. When called this method updates *finished_kernels* and *free_kernels* accordingly so that tasks dependent on this kernel can then be dispatched.
- `get_finished_tasks(self)` : Returns a list of finished tasks.

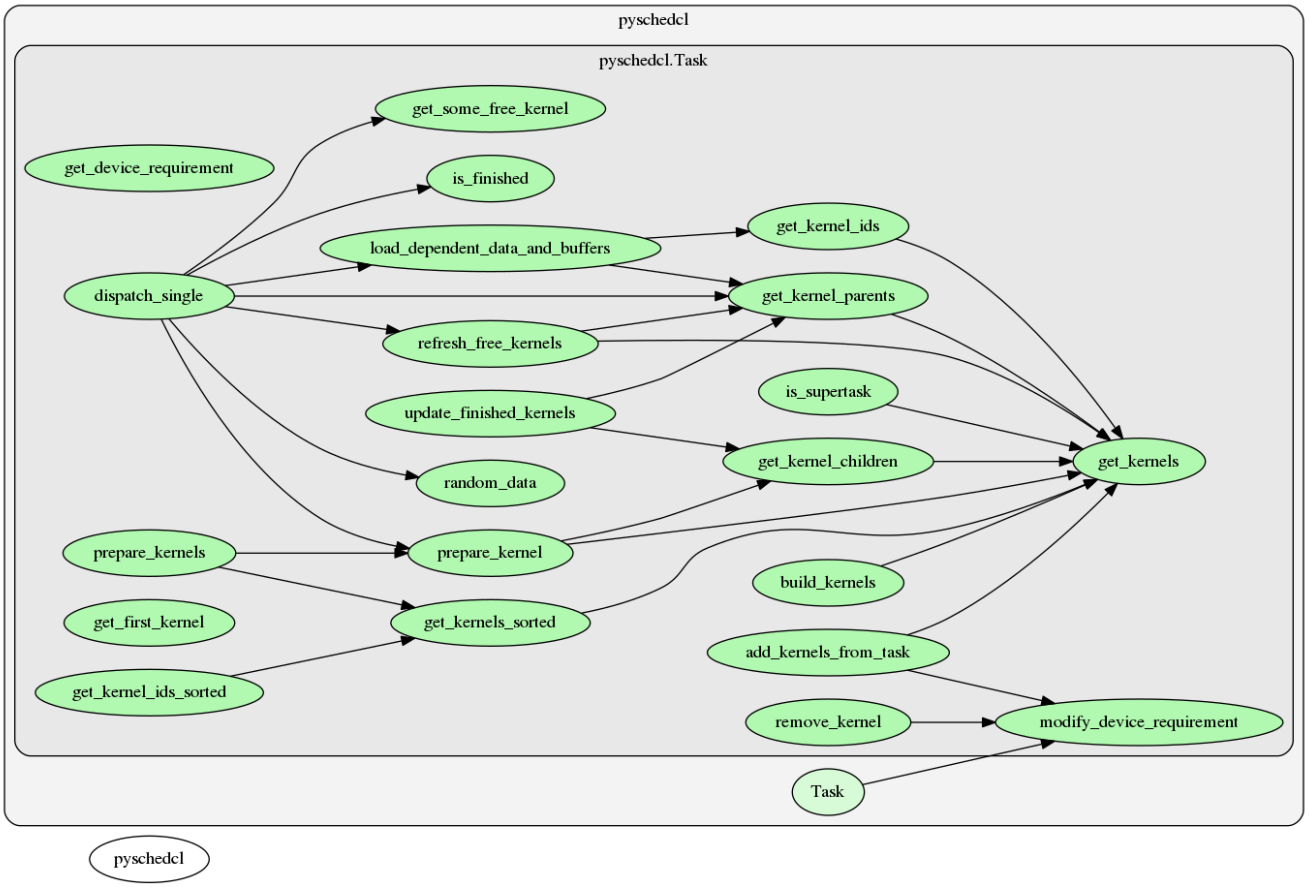

FIGURE 6.2: pyschedcl.Task Call Graph

- `update_finished_tasks(self, task)` : This method must be called whenever a task finishes execution. When called this method updates $finished\_tasks$ and $free\_tasks$ accordingly so that dependent tasks can then be dispatched.

- `kernel_data_transfer_size(self, kernel_r, kernel_s)` : Returns the total number of bytes of data that must be transferred between the given two kernels if there is a dependency between them.

- `task_data_transfer_size(self, task_r, task_s)` : Returns the total number of bytes of data that must be transferred between the given two tasks if there is a dependency between them.

- `get_free_kernels(self)` : Returns a list of kernels that don't have unmet dependencies.

- `get_kernel(self, kid)` : Returns a kernel given a kernel id.
- `get_kernel_parents(self, kid)` : Given a kernel id this method returns list of all the kernels that this kernel depends upon.
- `get_kernel_children(self, kid)` :Given a kernel id this method returns list of all the kernels that are dependent upon this kernel.
- `get_tasks(self)` : Returns a list of tasks
- `get_tasks_sorted(self)` : Returns a list of topologically sorted tasks.
- `get_all_task_dependencies(self)` : Returns dependencies between all tasks as a list of tuples.
- `get_task_parents(self, task)` : Given a task this method returns list of all the tasks that this task depends upon.
- `get_task_children(self, task)` : Given a task this method returns list of all the kernels that are dependent upon this task.
- `get_free_tasks(self)` : Returns a list of all tasks that have no unmet dependencies.
- `process_free_task(self)` : Removes a task from list of $free\_tasks$ and returns it.
- `merge_tasks(self, t1, t2)` : Merges tasks $t1$ and $t2$ into $t1$.
- `split_kernel_from_task(self, kernel, task)` : Removes the given kernel from the given task, creates a new task from that kernel and updates task dependencies accordingly. Returns the newly created task.
- `make_levels(G)` : Given a *networkx.DiGraph* returns a list of list of tasks, where each index contains a list of task that are at that level in the given *networkx.DiGraph*

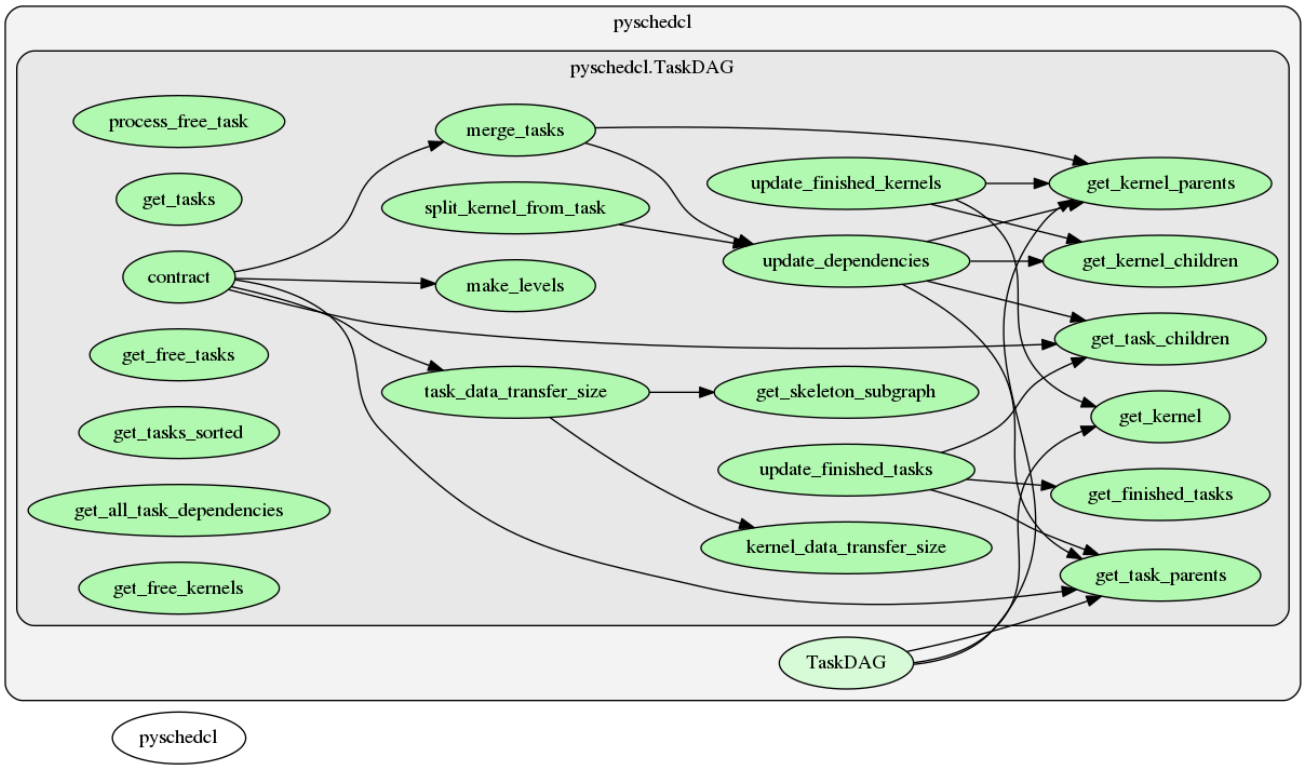

FIGURE 6.3: pyschedcl.TaskDAG Call Graph

- `contract(self)` : This is the contraction algorithm. Will be explained in detail later.

6.2.5 Miscellaneous Functions

- `convert_dtime(ts, dev_type)` : Converts given OpenCL device timestamp *ts* to corresponding host time *datetime.datetime*.
- `partition_round(elms, percent, exact=-1, total=100, *args, **kwargs)` : Partitions given dataset in a predctable way.
- `multiple_round(elms, percent, multiples, **kwargs)` : Partitions the given dataset such that the partitioned datasets are multiples of given number.
- `ctype(dtype)` : Converts a string datatype to corresponding *numpy* datatype.

- `make_ctype(dtype)` : This function returns an object that can be used to construct a vector datatype.
- `cs_logic(argument)` : Contains *Critical subSection* logic for recording events.
- `notify_callback(kernel, dev_type, dev_no, event_type, callback=blank_fn, *args, **kwargs)` : A wrapper function that generates and returns a callback function based on parameters. This callback function is run whenever a *enqueue* operation finsishes execution. User can optionally provide another callback which will run after completion $enqueue\_read\_buffers$ operation indicating completion of a kernel task.
- `generate_unique_id()` : Returns a universally unique identifier string.
- `get_platform(vendor_name)` : Returns $pyopencl.Platform$ given the vendor name.
- `get_multiple_devices(platform, dev_type, num_devs)` : Returns multiple *pyopencl.Device*s given a platform and device type.
- `get_single_device(platform, dev_type)` : Returns a $pyopencl.Device$ given a platform and device type.
- `get_sub_devices(platform, dev_type, num_devs, total_compute=16)` : Creates and returns sub devices given a platform and a device type.
- `create_command_queue_for_each(devs, ctx)` : Creates a $pyopencl.CommandQueue$ for each of the devices in the given $pyopencl.Context$.
- `host_initialize(num_gpus, num_cpus=2, local=False)` : : Given num_gpus, num_cpus this function will create the requested number of OpenCL devices, builds two OpenCL contexts and creates an OpenCL Command Queue for each device. Returns a tuple of cmd_qs, $ctxs$, $gpus$, $cpus$.

- `gantt_chart(kernels, title='Gantt Chart', bar_width=0.2, showgrid_x=False, showgrid_y=False, height=600, width=900)` : Given a list of Kernels it returns a *plotly* gantt chart depicting the execution timestamps of each kernel.

6.2.6 Global Variables and Constants

6.2.6.1 Global Variables

- `pyschedcl.numpy_types` : A dictionary containing *numpy* datatypes.
- `pyschedcl.nGPU, pyschedcl.nCPU` : Represents number of free devices.
- `pyschedcl.device_history` : A dictionary containing mapping between devices and Events
- `pyschedcl.ready_queue` : A queue containing available devices.
- `pyschedcl.cs` : This is the mutex used for implementing critical subsection logic.

6.2.6.2 Constants

- `pyschedcl.VEC_TYPES` : Contains a list of vector data types.
- `pyschedcl.CPU_FLOPS, pyschedcl.GPU_FLOPS` : Device FLOP specification.
- `pyschedcl.GPU_BANDWIDTH` : GPU bamdwidth specification
- `pyschedcl.MAX_GPU_ALLOC_SIZE, pyschedcl.MAX_CPU_ALLOC_SIZE` : Maximum memory that can be allocated on the device.
- `pyschedcl.gpu_offset, pyschedcl.cpu_offset` : Offset of OpenCL Device timestamp.

- `pyschedcl.gpu_prec, pyschedcl.cpu_prec` : Precision of OpenCL Device timestamp.

Chapter 7

Experimental Results

All the scheduling approaches discussed so far were evaluated on a heterogeneous system comprising four GPUs (NVIDIA Tesla C2050 clocked at 0.95 GHz) and two CPUs (Intel Xeon E5620 CPU clocked at 2.4 GHz).

7.1 Tasksets without dependencies

For evaluating the quality of each scheduling strategy, randomly generated task sets of sizes 10, 20 and 30 OpenCL kernels are used as input. It has been assume that all tasks arrive at the same time and the metric used for comparing different scheduling algorithms is the schedule makespan. They have been tested against

- the standard first come first serve strategy *FCFS* where each kernel is dispatched to any available device in the system.
- the strategy proposed in [5] where kernels are dispatched whole to either a CPU or a GPU. This strategy has been referred as *CPU-GPU*.
- a strategy where all kernels will be dispatched to either only CPU or only GPU.

The average span time for each of these scheduling algorithms is illustrated in Fig. 7.1. We observe that strategies where only devices of a single type are used do not

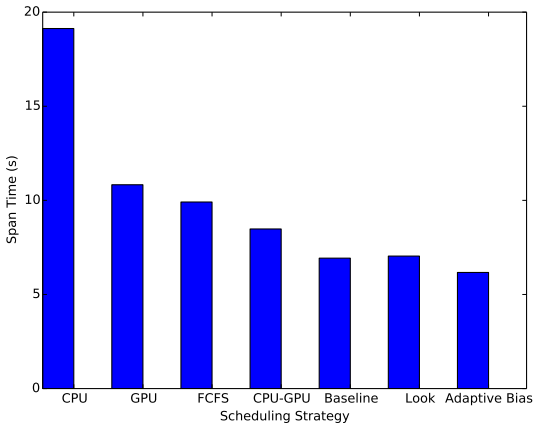

FIGURE 7.1: ML based scheduling framework

perform well, especially the strategy *CPU* where all kernels are dispatched only to the CPU device. This is followed by the strategy *GPU* and the standard *FCFS* algorithm. We observe that the strategy *CPU-GPU* which represents the work reported in [5] is outperformed by the three algorithms proposed in this paper indicating that partitioning should be used alongside scheduling for executing tasks on heterogeneous multicores.

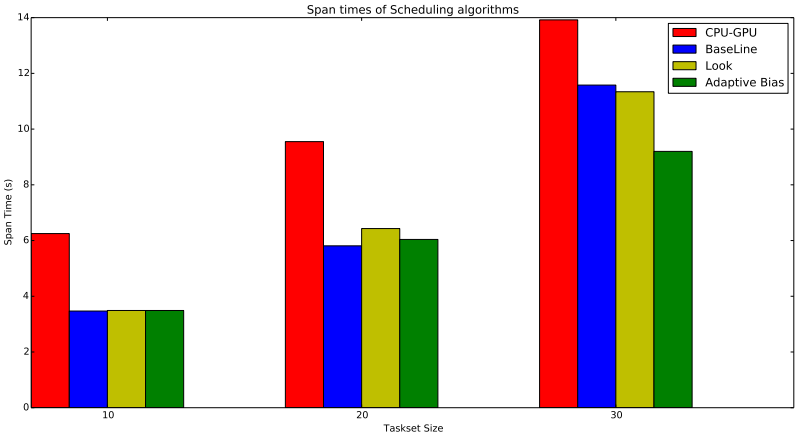

FIGURE 7.2: ML based scheduling framework

Fig. 7.2 shows a more detailed task set specific comparision between the ML based scheduling strategies in It can be observed that for each category of task sets, that parition-aware strategies outperform the strategy *CPU-GPU*. For task sets of size

10, it can be observed that the strategies Baseline, Lookahead and Adaptive Bias all perform similarly. For task sets of size 20, makespan for lookahead in comparison is slightly more than that of adaptive bias and baseline. For task sets of size 30, we observe adaptive bias performs the best followed by lookahead and baseline. All of the methods which involve partitioning outperform the method reported in [5]. The extent of partitioning workloads in each task set is different in each of the three methods. The baseline method ensures that optimal partition configurations for each task is maintained over the course of scheduling and in fact this method yields good results for task sets of size 10 and 20. The lookahead method compares two tasks at a time and decides whether additional resources should be sacrificed for executing a partitioned workload. This method performs well for larger task sets. The best results are obtained using adaptive bias. Tasks are categorized beforehand based on the bias towards a particular device. Application to device mapping is decided between tasks belonging to different task queues based on device bias and with the help of the static computation overhead measure. The results obtained using this method is close to the the other two methods for task sets of sizes 10 and 20 and significantly better for task sets of size 30.

7.2 Tasksets with dependencies

Fig. 7.3 and Fig. 7.4 show the profile of a DAG containing and Covariance Kernel and Correlation Kernel where Correlation Kernel depends on Covariance. Similarly figures Fig. 7.5 and Fig. 7.6 show the profile of a DAG containing and Reduce Kernel and Mean Kernel where Mean Kernel depends on Reduce Kernel. In both the cases there is a slight decrease in overall makespan time when buffers are reused. (i.e when Contraction Algorithm is applied.) However significant improvement of make span hasn't been observed on all cases eventhough we have eliminated the data transfer overhead. The reason is that the dependency scheduling algorithm waits for the callback to update the queues, and the callback trigger is not completely

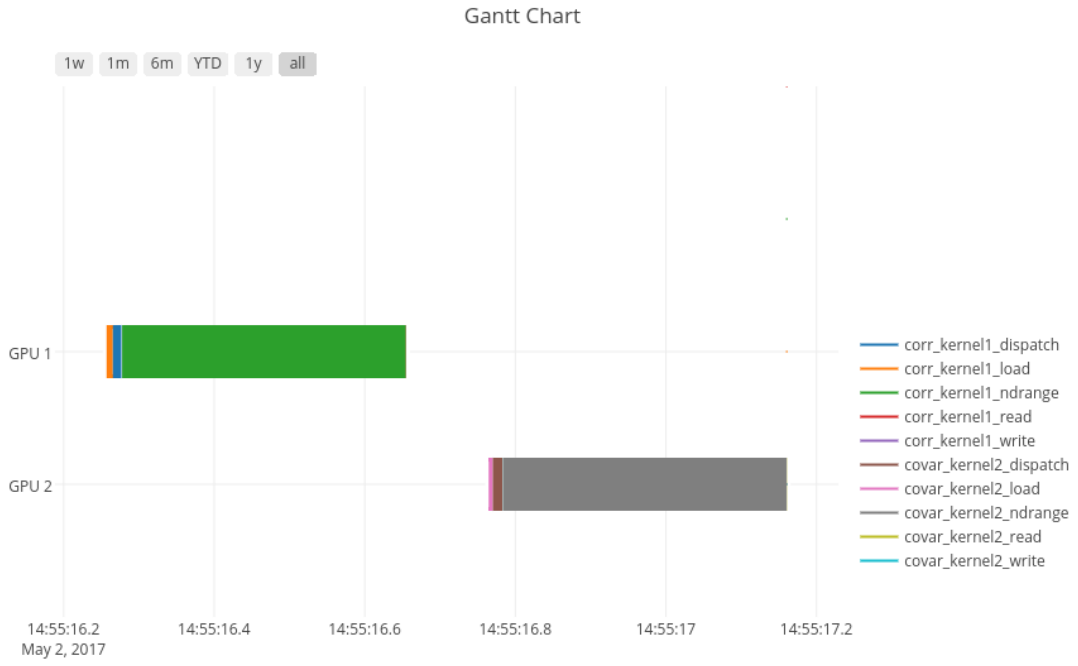

FIGURE 7.3: : Covariance and Correlation: DAG schedule without buffer reuse.

reliable. This cancels the improvements achieved as even minor delays in callback trigger incurs significant stall to the schedule as OpenCL tasks are pretty fast. This is the reason why OpenCL device seems to idle for a significant time. During this time the framework is either waiting for fresh tasks to arrive or preparing tasks for dispatcher.

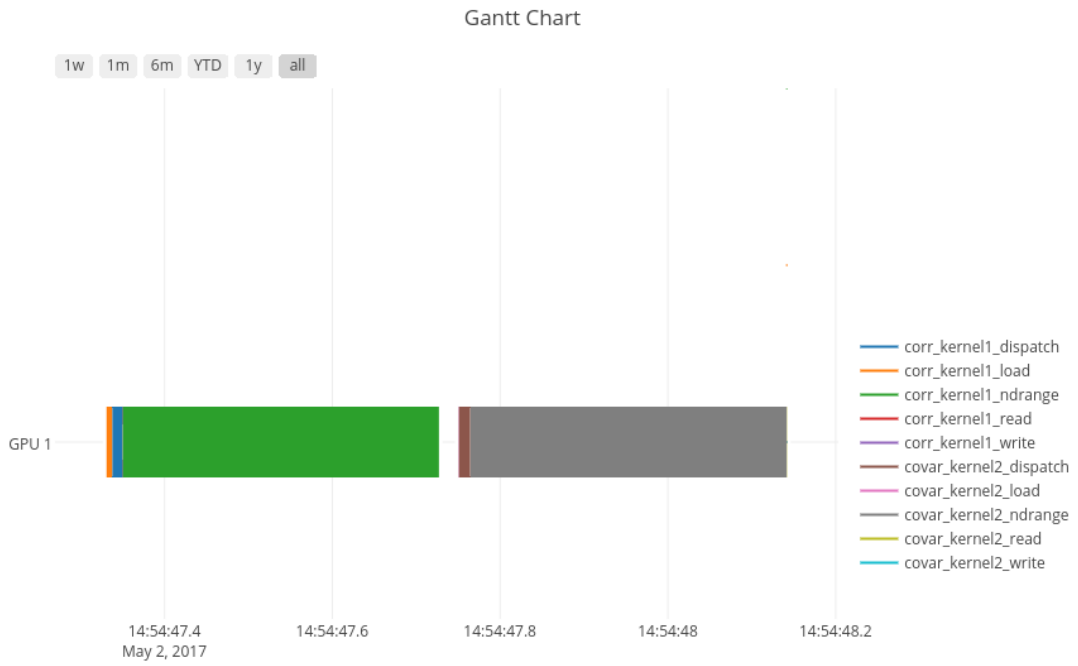

FIGURE 7.4: : Covariance and Correlation: DAG schedule with buffer reuse.

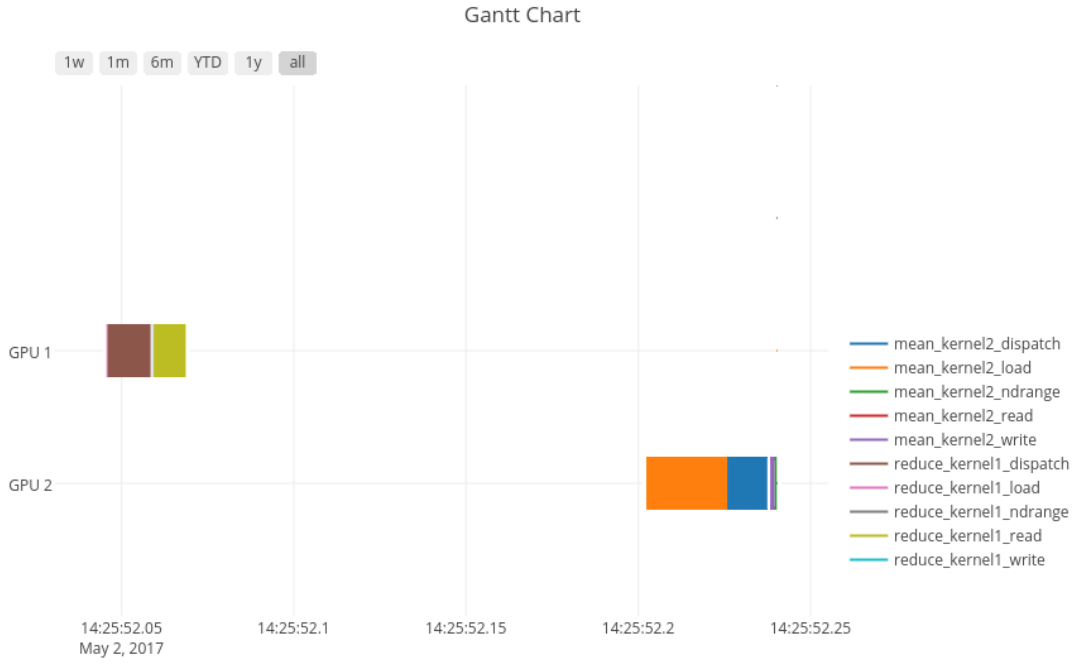

FIGURE 7.5

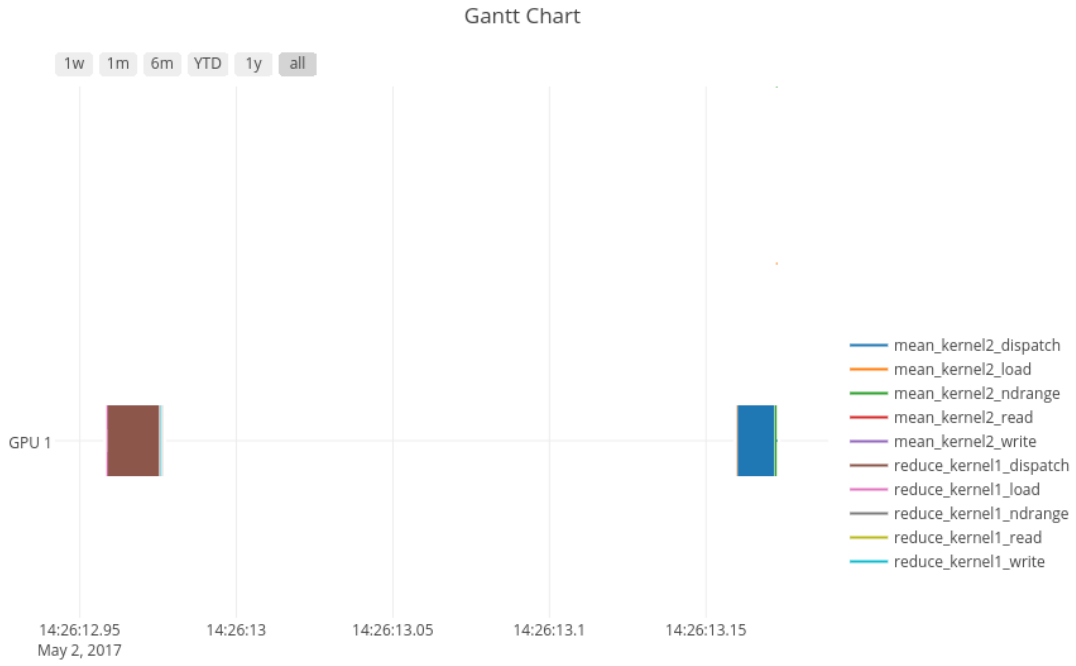
Gantt Chart
1w
1m
6m
YTD
1y
all
GPU 1
mean_kernel2_dispatch
mean_kernel2_load
mean_kernel2_ndrange
mean_kernel2_read
mean_kernel2_write
reduce_kernel1_dispatch
reduce_kernel1_load
reduce_kernel1_ndrange
reduce_kernel1_read
reduce_kernel1_write
14:26:12.95
May 2, 2017
14:26:13
14:26:13.05
14:26:13.1
14:26:13.15

FIGURE 7.6

Chapter 8

Conclusions

The present work reports a partition aware heterogeneous scheduling framework for OpenCL programs which can automatically handle the dependencies between programs. Three partition aware scheduling algorithms for independent tasksets have been implemented on this framework which outperform the current state of the art methods with no support for runtime program partitioning. We have seen how framework can can handle tasks with dependencies. The discussed strategy for improving schedules with dependencies has been implemented and results have been demonstrated.

8.1 Scope of Future Work

- The framework can be extended to handle tasks with different arrival times and strategies for scheduling dynamic tasksets that vary at runtime can be explored.
- Scope of applying Round-Robin like scheduling strategies to OpenCL tasks can be explored.

- This framework can be developed into a full-fledged scheduler program which runs in the background as a server so that tasks can be submitted to this server using HTTP REST API.

- Static Program Analysis can be used to automate generation of the major part of the Kernel Specification Files. Thus a graphic interface can be made to assist users in writing the Kernel Specification files.

Bibliography

[1] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[2] Plotly Technologies Inc. Collaborative data science, 2015.

[3] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[4] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[5] Zheng Wang Yuan Wen and Michael O'Boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *HIPC*, 2014.