PySchedCL: A Framework for Automatically Exploiting Concurrency in Heterogeneous Data-Parallel Applications

M.Tech Project - II Thesis report submitted to

Indian Institute of Technology Kharagpur

in fulfilment for the award of the degree of

Dual Degree (B.Tech + M.Tech)

in

Computer Science and Engineering

by

Siddharth Singh

(15CS30032)

Under the supervision of

Professor Soumyajit Dey

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

Spring Semester, 2019-20

June 2, 2020

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: June 2, 2020 (Siddharth Singh)

Place: Kharagpur (15CS30032)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

KHARAGPUR - 721302, INDIA

CERTIFICATE

This is to certify that the project report entitled "**PySchedCL: A Framework for Automatically Exploiting Concurrency in Heterogeneous Data-Parallel Applications**" submitted by **Siddharth Singh** (Roll No. 15CS30032) to Indian Institute of Technology Kharagpur towards partial fulfilment of requirements for the award of degree of Dual Degree (B.Tech + M.Tech) in Computer Science and Engineering is a record of bona fide work carried out by him under my supervision and guidance during Spring Semester, 2019-20.

Date: June 2, 2020

Place: Kharagpur

Professor Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India

Abstract

Name of the student: **Siddharth Singh** Roll No: **15CS30032**

Degree for which submitted: **Dual Degree (B.Tech + M.Tech)**

Department: **Department of Computer Science and Engineering**

Thesis title: **PySchedCL: A Framework for Automatically Exploiting Concurrency in Heterogeneous Data-Parallel Applications**

Thesis supervisor: **Professor Soumyajit Dey**

Month and year of thesis submission: **June 2, 2020**

In the past decade, high performance compute capabilities exhibited by heterogeneous GPGPU platforms have led to the popularity of data parallel programming languages such as CUDA and OpenCL. Such languages, however, involve a steep learning curve as well as developing an extensive understanding of the underlying architecture of the compute devices in heterogeneous platforms. This has led to the emergence of several High Performance Computing frameworks which provide high-level abstractions for easing the development of data-parallel applications on heterogeneous platforms. However, the scheduling decisions undertaken by such frameworks do not sufficiently exploit the concurrency inherent in a data parallel application to its full potential. We propose a framework called *PySchedCL*, whose design philosophy is along similar lines as that of other HPC frameworks, with a specific focus on exploring fine-grained concurrency aware scheduling decisions that completely harness the power of heterogeneous CPU/GPU architectures. We showcase the efficacy of such scheduling decisions over popular dynamic scheduling

schemes by conducting extensive experimental evaluations for a Machine Learning based inferencing application. We also experiment with automated coarse-grained scheduling algorithm that rely on Machine learning to schedule heterogenous applications.

Acknowledgements

I am deeply grateful to my supervisor **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

Chapter 1

Introduction

1.1 Introduction

The rise of parallel programming languages like OpenCL [17], CUDA [15] and widespread availability of heterogeneous computing platforms comprising CPUs and GPUs have paved the way for researchers to develop efficient scientific computing workloads spanning across diverse domains of science. In the past few years, both of these heterogeneous programming languages have been extensively used for developing high performance computing (HPC) applications for execution on heterogeneous multicore architectures ranging from cluster level workstations to heterogeneous embedded platforms comprising multiple CPUs and GPUs. Both frameworks support asynchronous event driven programming models that enable both data parallel and task parallel paradigms of computation for implementing high performance parallel applications. The data parallel programming model has provisions for implementing a computational kernel which represents the core computation for a given algorithm. A data parallel computational kernel launches multiple threads in parallel across multiple SIMD enabled compute units. Each thread applies the specified kernel transformation to designated data points of the input data space. The task parallel programming model supports parallelism at the task/kernel level where application task graphs comprising multiple kernels with dependencies, each representing distinct computational transformations can be dispatched and executed on multiple

devices in a target heterogeneous platform. The OpenCL runtime system additionally has provision for program portability across different types of devices i.e. the same computational kernel source code can be compiled into device specific binaries for execution on different devices.

Given heterogeneous platforms comprising multiple devices of varying computational power, determining efficient architecture-to-application mapping decisions require extensive domain knowledge of platform level characteristics as well as precedence constraints enforced by the application task graph. As an illustrative example, let us consider a simple fork-join task graph in Figure 1.1.

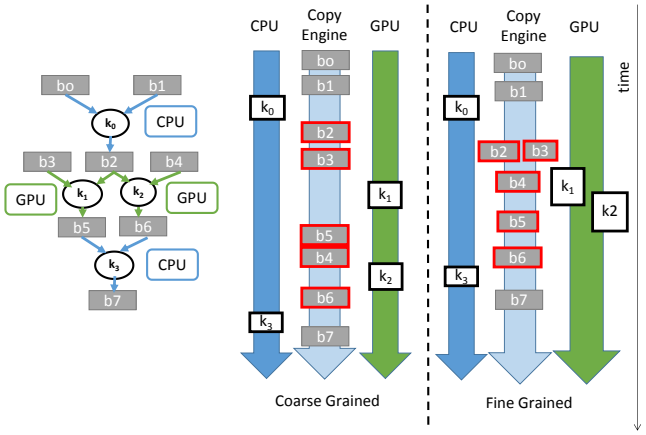

FIGURE 1.1: DAG Mapping Decisions

We consider a heterogeneous platform comprising a single CPU and a single GPU along with a DMA copy engine responsible for transferring data across the PCI-Express bus from the CPU to the GPU and back. The fork-join graph comprises four tasks, each representing some computational kernel which takes as input two input buffers and produces one output buffer. In Fig. 1.1, the rectangular nodes represent input output buffers and the circular nodes represent kernels. We use this convention throughout the paper. The edges between a buffer and task represents the precedence constraints between tasks as well. Given a heterogeneous compute platform comprising a single CPU and a single GPU there can exist a total of 16 task-device mappings for this task graph where task(s) are either mapped to a GPU device or a CPU device. In Fig. 1.1, we explore one of the 16 possible mappings where k_0 and k_3 are mapped to a CPU device, k_1 and k_2 are mapped to a GPU device.

Scheduling decisions for general application DAGs are coarse-grained in the sense that each task is mapped to a single device at a time and the associated kernel execution, buffer reads and writes are finished completely before proceeding to execute successors of the kernels. In Fig. 1.1, for kernel k_2 to start execution, k_1 must finish and the copy engine should copy the resultant buffer b_5 to the host. After that the required input buffer b_4 has to be copied to the GPU device. The scheduling decisions are achieved by designing complex host programs that orchestrate the process of mapping individual kernels to target devices of the heterogeneous platform while maintaining precedence constraints. Alternatively, there exist several frameworks proposed in the recent past that alleviate the burden of implementing such complex orchestrators for undertaking coarse-grained scheduling decisions. The frameworks can be classified into two broad categories - i) frameworks like [11, 8] that provide either a top-level API or additional programming constructs using which an end designer has to modify existing OpenCL benchmark source code and ii) frameworks such as StarPU, MultiCL that provides scheduling engines [2, 1] optimized for heterogeneous clusters with support for custom scheduling heuristics for mapping a dataflow graph of OpenCL kernels on a heterogeneous platform. Both styles rely on deriving coarse-grained scheduling decisions for application DAGs.

In contrast, we believe scheduling decisions should be more fine-grained in nature allowing execution of multiple tasks in the same device and interleaving copy operations with execute operations. This is exemplified in the right hand side scheduling option of Fig. 1.1. We can observe for kernel k_1, the two input buffers b_2 and b_3 can be transferred by the copy engine in parallel. Also while kernel k_1 is executed, b_4 can be transferred asynchronously to the GPU device. The kernel k_2 executes in parallel with k_1 while sharing the same GPU resource. As a result, we observe that the individual times of k_1 and k_2 increase. However, the overall time to finish DAG execution decreases.

Implementing such fine-grained scheduling requires designing an even more complex host program capable of i) asynchronously interleaving data transfers as and when required and ii) clustering multiple tasks to the same device as and when feasible. These scheduling decisions can be achieved by setting up multiple worker queues per device and asynchronously enqueueing commands for executing multiple kernels on the same device. For the CUDA runtime system, these worker queues are referred

as *CUDA streams*. For the OpenCL runtime these are referred as *command queues*. Naturally, the end user has to consider the computational capability of the device and the individual computational requirements of each concurrent kernel before dispatch.

We propose *PySchedCL* a platform agnostic programming framework which is possibly the first computer-aided design solution that is capable of automating the process of deriving both coarse-grained and fine-grained scheduling decisions for efficient collaborative execution of application task graphs on heterogeneous multicores comprising CPU and GPU devices. The proposed framework supports reduction of considerable implementation overhead and automatically outputs scheduling decisions that exploit concurrency with minimal intervention by the programmer. The framework is built using the widely used PyOpenCL API [13] and facilitates rapid development and deployment of OpenCL applications. We choose OpenCL, since it offers device portability, thus supporting a myriad of heterogeneous compute devices such as CPU, GPUs, FPGAs, DSPs etc. Our framework enables the user to concentrate only on developing OpenCL kernels and providing minimum guidance parameters that would help in finally determining near optimal runtime scheduling decisions for data parallel applications on a target heterogeneous CPU-GPU platform. We note the optimizations proposed are generic and the ideas can be leveraged for any data parallel heterogeneous setting. The salient features of the proposed framework are enumerated as follows.

- The framework supports a design frontend that will facilitate programmers to develop and execute application task graphs without having to consider the intricacies of runtime environment. The frontend has provisions for a specification file using which the programmer can construct the input specification for each OpenCL kernel in the task graph as well as provide precedence constraints. The task of the application designer is only to develop individual kernels and populate these specification files.
- The framework supports a scheduling engine which automatically issues directives required by the OpenCL runtime system for executing an application comprising either a single kernel or multiple kernels with dependencies efficiently. This completely bypasses the the requirement of manually writing a

host program which captures all low level scheduling decisions. The scheduling engine mimics the behaviour of the orchestrating host program and has support for clustering kernels in a task graph as well as automatically making decisions that involve concurrent kernel execution on the same device.

- In addition to the modules above, the framework has provisions for specifying guidance parameters using which the framework can automatically set up device worker queues that can exploit application level concurrency on heterogeneous compute platforms. We showcase the efficacy of this approach by considering a inference pipeline for the Transformer Nerual Network architecture [18] which is an application that provides ample scope for concurrency and parallelization. We provide extensive experimental results for the same comparing our approaches with standard dynamic list scheduling algorithms provided in frameworks such as StarPU [2], SOCL [8] etc. The static fine-grained scheduling approach exhibits speedups in the range of $1.4 - 3.4x$ when compared to the dynamic coarse-grained scheduling approaches.

The framework thus eases complex OpenCL application development with the help of specification files and automatically exploits available fine/coarse grained scheduling techniques for mapping data-computational kernels to OpenCL compliant devices in an heterogeneous CPU-GPU platform. In the remainder of the paper, we first present necessary background and domain knowledge of the OpenCL runtime system (Section **??**), support offered by existing frameworks and subsequently motivation for designing our proposed framework. This is followed by the software architecture and the details of the various components of the framework in Section **??**. We perform extensive experimentation and provide a comparative evaluation between existing scheduling approaches in Section **??**.

Chapter 2

OpenCL Background and Related Work

2.1 OpenCL Background

Any OpenCL application typically comprises two distinct program entities - i) the *host* which is a single threaded sequential program executing on one CPU core that orchestrates the entire process of managing data and issuing directives for parallel execution, and ii) kernel(s) which execute on devices with support for vector processing (CPU,GPU,FPGA,DSP). For every computational kernel, the single-threaded host program leverages command queues supported by the OpenCL API to issue commands for performing the following operations - i) copying the data from host to input buffers resident on device memory (Host to Device or H2D transfer), ii) launching multiple instances of the same kernel to process the data copied to the device and iii) copying back the data stored in output buffers in the device after the kernel has finished processing back to the host memory (Device to Host or D2H transfer).

As an illustrative example, we consider a simple OpenCL application which performs a vector addition followed by an element-wise trigonometric sine operation. The vector addition kernel *vadd* executes on device GPU_0. It takes as input two input buffers ($b0$ and $b1$) performs element-wise addition and produces an output

buffer ($b2$). The kernel *vsin* executes on device GPU_1 and takes one buffer ($b3$) and performs an inplace element-wise sine operation. In Fig. 2.1, the OpenCL host program sets up command queues for each GPU device. For GPU_0, the host first issues two *write* commands (`clEnqueueWrite()`) for buffers $b0$ and $b1$ followed by a *barrier* directive (`clEnqueueBarrier()`). The *barrier* command in general ensures that all commands enqueued previously finish before proceeding to execute commands enqueued after the barrier. In this case, it is ensured that the write commands are finished before processing the next command in the queue. The host next issues one execute command (`clEnqueueNDRangeKernel()`)followed by a barrier directive and finally one read command (`clEnqueueReadBuffer()`) followed again by a barrier directive. We note all the functions with the `clEnqueue` prefix asynchronously issues these commands to the OpenCL runtime system i.e. the host does not have to explicitly wait for a particular command to finish. It simply enqueues the commands and is free to execute something else while those commands are executed on the target device.

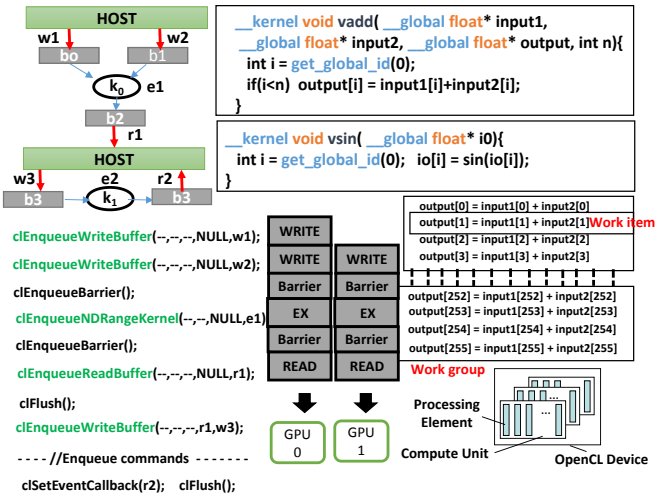

FIGURE 2.1: OpenCL Execution

The command `clEnqueueNDRangeKernel()` spawns a collection of threads referred as *work items* where each work item executes on a *processing element* on the heterogeneous platform. Each work item is referred by a unique identifier i obtained using the `get_global_id()` OpenCL function and is responsible for the addition of data points in the two input buffers $b0$ and $b1$ (`input1[i]` and `input2[i]` in function call) and storing the result in the corresponding location of the output buffer (`output[i]`

in function call). Work items are further grouped into *work groups* and each work group is scheduled for execution on a *compute unit* in an OpenCL compliant device. A *compute unit* may be a Symmetric Multiprocessor(SM) for a GPU device, a single core of a multicore CPU etc.

Similarly, it may be observed from Fig. 2.1, the host issues a write command (for buffer $b3$), the execute command for *vsin* kernel and one read command (for buffer $b3$ to the command queue pertaining to device GPU_1. We note that the write command for *vsin* can start execution once the read command for the *vadd* has finished. The OpenCL runtime system has provisional APIs for specifying dependencies between commands. This is done using *events* which are objects that communicate the status of commands issued from the command queue. These events can be used for i) monitoring the execution of read/write operations and kernel execution, ii) enforcing dependencies across multiple commands and iii) notifying the host program about the completion of a command on the device.

In Fig. 2.1, each *write*, *ndrange* and *read* command c enqueued is associated with an event ev. This is specified in the last argument of each command c with `clEnqueue` prefix. The second last argument of the function represents events on which the event ev associated with command c is dependent for execution. For our representative example, it may be observed that the event $w3$ is dependent on $r1$. This is specified in the `clEnqueueWriteBuffer` command for buffer $w3$. This is followed by the other enqueue OpenCL function calls such as barriers, launching the kernel and reading the final output buffer for *vsin*.

Finally, for the event associated with the last read command in the application (associated with event $r2$ in Fig. 2.1), a *callback function* is registered which is responsible for notifying the host when the computation has finished processing on a device. We note the `clEnqueue` OpenCL functions merely enqueue operations to each command queue for the devices. After enqueuing all commands to a command queue, the function `clFlush()` is invoked which informs the OpenCL runtime system to start dispatching the commands pushed to the command queue for execution.

OpenCL events are thus particularly useful for enforcing read/write and execute dependencies between multiple kernels in application task graphs which are typically represented by a directed acyclic graph (DAG) of OpenCL kernels. We next discuss

a slightly more complex DAG example and discuss how coarse-grained and fine-grained scheduling decisions result in different command queue configurations in the following subsection.

2.2 Related Work

Given the rich API support by both heterogeneous programming models CUDA and OpenCL, several frameworks have emerged over the last few years with the objective of providing user friendly solutions for development of data parallel applications. Frameworks such as OpenACC [9] supports a directive based programming model where relevant annotations in sequential C programs generate data parallel CUDA code for execution on the GPU. Similarly, the framework HiCUDA [7] is another high level directive based programming language for generating CUDA binaries from sequential C source code. Frameworks such as GMAC [3] offer a data centric programming model relieving the end designer from making explicit memory requests while implementing applications. We note that the frameworks discussed are built using the CUDA API and are restricted for use on heterogeneous systems with NVIDIA GPU architectures only. In constrast several OpenCL based frameworks have also been envisioned in the past decade for general purpose heterogeneous programming.

The most notable framework offering high-level abstractions for developing OpenCL applications is SkelCL [16] which offers algorithmic skeletons for developing data-parallel programs for execution across multiple GPUs. In this context, skeletons refer to higher order functions such as map, reduce, scan etc. which can be leveraged for implementing data parallel algorithms. Note, the primary approach of this work is complementary in the sense that they focus on rapid kernel development , while our work focuses on scheduling optimizations on target heterogeneous architectures. The VirtCL framework [21] provides an abstraction layer between the programmer and the OpenCL runtime system acting as a hypervisor for scheduling multiple OpenCL applications. The abstraction framework leverages a profile driven history based scheduling scheme for dispatching OpenCL kernels on multiple devices. However a major limitation for VirtCL is that it cannot operate with devices belonging to

different platforms. Our framework in contrast is suited to work with different OpenCL platforms and opts for a static based scheduling approach for mapping OpenCL kernels. There also exists frameworks such as SnUCL [12], VOCL [20], MultiCL [1] etc. that extend upon the OpenCL runtime API that allow OpenCL applications to leverage devices belonging to heterogeneous clusters. This work also presents a unified OpenCL implementation by incorporating a task queuing extension layer. However, such APIs still require explicit kernel and host program development and lack support for intelligent scheduling techniques mentioned earlier.

There also exist unified scheduling frameworks for heterogeneous platforms such as StarPU [2] which provides users an interface for designing and experimenting scheduling policies for both CUDA and OpenCL applications. The StarPU runtime system allows users to design scheduling priority functions for experimentation. An extension on StarPU [10] supports scheduling multiple tasks in parallel on a heterogeneous system. The work reported in [8] presents an unified OpenCL implementation called SOCL which directly extends upon StarPU for handling and managing execution of OpenCL workloads across multiple devices using different scheduling policies. The SOCL API typically supports scheduling algorithms that require profiling information for each task on each device in the system beforehand. The most recent work in this domain is reported in [11], which proposes a set of APIs on top of OpenCL using which dependencies can be specified for application DAGs.

Despite the vast number of frameworks available, our proposed framework relies on specification files using which programmers can bypass the overhead of implementing complex host programs and design data parallel applications with ease. We also present a novel software architecture that is capable of automatically extracting both application level and platform level concurrency. Currently the scheduling schemes lack support for performance models necessary for obtaining near-optimal schedules. Future work entails investigating machine learning assisted techniques [5, 6, 14, 19, 4] for the same. We believe the robust API support in our framework would allow researchers to investigate these avenues and accordingly design and validate novel scheduling algorithms for heterogeneous platforms.

Chapter 3

Problem Formulation

3.1 Motivation

We consider a transformer application [18] which is a popular Deep Learning Neural Network pipeline for Natural Language Processing (NLP) tasks. The application exhibits ample scopes for exploiting concurrency with the possibility of executing multiple instances of standard General Matrix Multiply (GEMM) kernels in parallel. A sample DAG for one layer of the transformer network is presented in Fig. 3.1. We shall discuss necessary background and details about the workload later in the Experimental Results section.

For the purpose of our motivation, we present a level wise view of the DAG under consideration in the left hand side of Fig. 3.1, which contains a total of 8 kernels. As per our earlier convention, the rectangular nodes represent input output buffers and the circular nodes represent kernels Each kernel is labeled with the corresponding level number starting from 1. Initially there is a copy operation which copies the same buffer to each of the kernels at level 1. Each of the kernels in levels 1,4,5,6 represent General Matrix Multiply (GEMM) kernels where each kernel takes as input two buffers and produces one output buffer. The kernels in level 2 and level 3 represent transpose and softmax operations respectively, each processing one input buffer to produce one output buffer. The edges between rectangular nodes i.e. buffers represent data dependencies for the DAG. For enforcing precedence constraints between

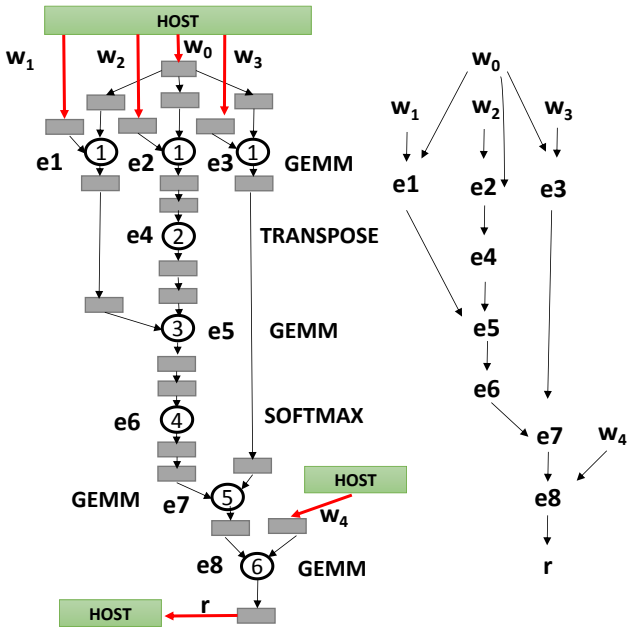

FIGURE 3.1: Event Dependencies for DAG

any pair of kernels (k_i, k_j), a programmer shall set event dependencies between read commands for output buffers of k_i and write commands for input buffers of k_j, as was observed in Fig. 2.1. For our transformer DAG depicted in Fig. 3.1, the programmer is required to set event dependencies between read commands for output buffers of kernels in level i and write commands for input buffers of kernels in level $i+1$. If the entire DAG was mapped to a single GPU device, explicit reads and writes for dependent buffers between kernels in levels 1-5 are not required. For this case, the programmer needs to set up event dependencies between *ndrange* commands of kernels in levels i and $i+1$.

In the left hand side of Fig 3. we label each kernel k of the DAG with event e_k associated with the corresponding *ndrange* command for that kernel. Apart from this, we have a *write* command w_0 responsible for copying one common buffer to be used for each GEMM kernel in level 1. We also have *write* commands w_1, w_2, w_3 for each of the remaining buffers required by GEMM kernels in level 1 and a *write* command w_4 for a buffer required by GEMM kernel in level 6. Finally we have a *read* command r for the output buffer of the GEMM kernel in level 6. The dependencies between these events are depicted in the corresponding event dependency graph in the right hand side of Fig. 3.1. The end designer is burdened with the task of manually writing a host program that will capture the event dependencies illustrated

in this dependency graph for ensuring that precedence relations of the DAG are met during execution. This is achieved by using the complex programming constructs for OpenCL events and callback functions as discussed earlier.

We next examine how coarse-grained and fine-grained scheduling decisions are made for mapping this DAG onto a single GPU device with the help of Fig. 3.2.

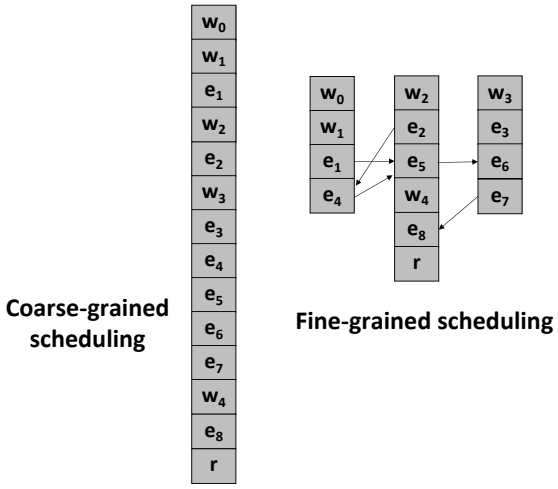

FIGURE 3.2: Command Queue Configurations for Scheduling

Coarse grained scheduling decisions refer to the context when all operations of kernels of a DAG are executed in its entirety first. This is achieved by setting up a single command queue on the GPU device. As a consequence, we can observe that all commands labelled by the events used in Fig. 3.1 execute serially on the GPU device. In contrast if we set up multiple command queues, there is a possibility of leveraging fine grained scheduling decisions that can interleave data transfers with *ndrange* operations and can execute multiple ndrange operations concurrently. In Fig. 3.2, we setup 3 command queues for achieving this. One can observe that the writes w_0, w_2, w_3 can now happen simultaneously. The ndrange operations e_1, e_2 and e_3 can also execute concurrently on the same GPU device. However, the other commands, despite belonging to different command queues would not be able to execute simultaneously due to the precedence relationships enforced by the event dependency graph illustrated in of Fig. 3.1. These dependencies are represented by inter-queue edges between events in Fig. 3.2.

We note that both the cases represented in Fig. 3.2 depict one of the possible command queue configurations. In our representative example, for coarse-grained

scheduling, one can have another command queue configuration where the *write* command associated with w_1 is enqueued before that of w_0 or where the *write* command for w_4 is enqueued anywhere before the *ndrange* command for e_8. In a similar vein, one can have different command queue configurations for fine-grained scheduling as well. We next analyze how coarse-grained and fine-grained scheduling decisions for executing this DAG on a single GPU device compare with the help of the Gantt charts in Figs 3.3 and 3.4.

We execute the DAG on a heterogeneous platform comprising an NVIDIA GTX-970 GPU device and a Quadcore Intel i5-4690K CPU device. The Gantt chart in Fig 3.1 represents the case where a single command queue is set up for the GPU device and all the *read*, *write* and *ndrange* commands for each of the 8 kernels are enqueued. The x-axis of the Gantt chart represents time in milliseconds (ms). The y-axis of the Gantt chart represents the kernels constituting the DAG. Each kernel is labelled with the level of the DAG to which it belongs followed by the name of the kernel operation and *write*, *read* and *ndrange* commands pertaining to it. Each green rectangle denotes the time taken by a *write* command. Each read and brown rectangle denotes the time taken by *ndrange* and *read* commands respectively. As evident from the corresponding Gantt chart, each command associated with each kernel executes one at a time on the GPU device resulting in an execution time of 105ms. We observe that writes occur for the first copy operation (associated with event $w0$) and for buffers of kernels in level 1 and the kernel in level 6. For GEMM kernels in level 1, data is transferred for input buffers associated with events w_1, w_2 and w_3. Similarly for the GEMM kernel in level 6, data is transferred for the input buffer associated with event $w4$. Finally the last read associated with event r occurs for the kernel in level 6. We note that each of the operations are executed sequentially mapped to a single GPU device using a single command queue.

In contrast, if we set up three command queues for the same GPU device and dispatch our kernels intelligently, we observe an 8% reduction in execution time , with the DAG finishing in 95ms. This reduction in time maybe attributed to the interleaving of data transfers (*write* commands associated with $w1, w2$ and $w3$) with execute operations (*ndrange* commands associated with events $e1, e2, e3$) for kernels in levels 1. It can be observed that while *ndrange* command associated with $e1$ is executing, the buffer associated with $w2$ can be copied simultaneously. Similarly,

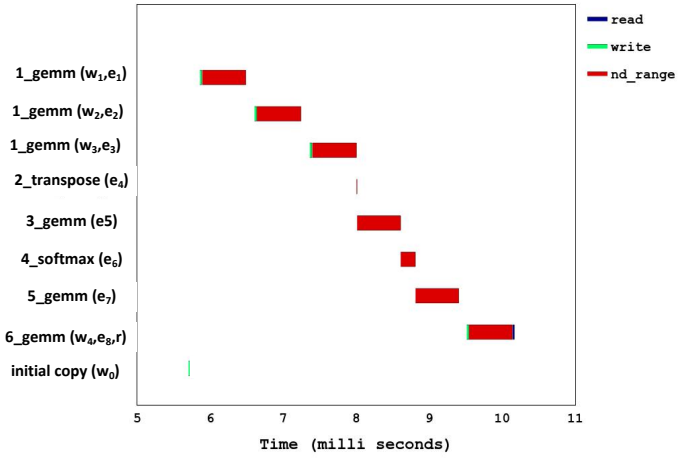

FIGURE 3.3: Sequential Execution on GPU device

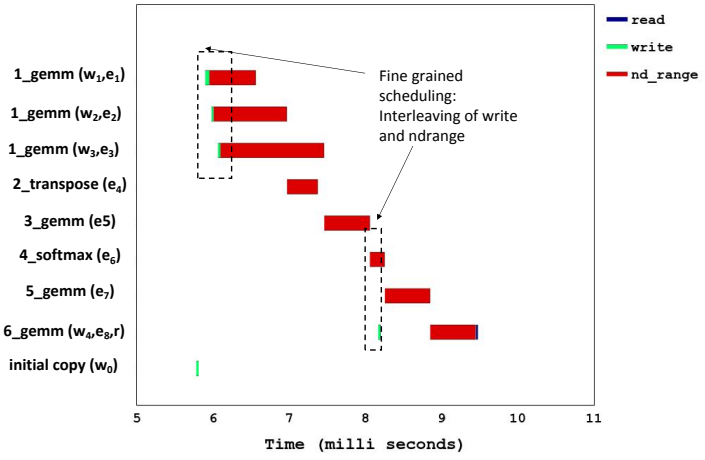

FIGURE 3.4: Concurrent Execution on GPU device

$w3$ can also be copied while $e1$ and $e2$ are executing. Additionally, it can be seen that all kernels in level 1 are executing concurrently for the same device. It may be further observed, the individual execution times for each kernel increases slightly as a result. This is due to the fact, that different work groups of different kernels that have been concurrently dispatched are scheduled in a round robin fashion to the compute units of the device, thus causing resource contention. However, the total time for finishing both kernels concurrently is lesser than the case when they are dispatched in sequence. We note that concurrency between three kernels and three data transfers yields a decrease of 8 % in execution time. In general, for one layer of the transformer a maximum of 16 such DAGs can run in parallel. In such a setting where there is more concurrency to exploit, fine grained scheduling that exploits

both the CPU and the GPU devices of the heterogeneous platform by setting up multiple command queues per device shall yield more speedups.

The typical dispatch mechanisms offered by list scheduling heuristics available in SOCL, StarPU and MultiCL are optimized for heterogeneous clusters comprising multiple devices and rely on coarse-grained scheduling decisions. They do not leverage the benefits obtained by using fine-grained scheduling decisions. We implement *PySchedCL* as a scheduling framework optimized for heterogeneous multicore platforms where devices support concurrent execution. Provided with the right guidance parameters by the designer, the framework shall automatically produce efficient data-parallel mapping solutions that can exploit concurrency both at the application and at the platform level.

3.2 Formal Problem Statement

Let us consider a heterogeneous platform $\mathcal{P}$ depicted in Fig. 3.5 which comprises a CPU device and a GPU device connected via a PCI-Express bus. Each device has support for executing multiple kernels simultaneously. The OpenCL standard supports device fission for CPU devices i.e. a single CPU device can be partitioned into multiple subdevices, thereby enabling concurrent execution for the same. We consider as GPU an NVIDIA device with Hyper-Q support [15]. Hyper-Q offers a solution that allows the CPU host to dispatch multiple kernels simultaneously on the GPU device with the help of hardware managed work queues.

Let us represent an OpenCL application graph as a directed acyclic graph (DAG) $G = \langle K, B, E_I, E_O, E \rangle$ where K denotes the set of OpenCL kernels, $B = B_I \bigcup B_O$ represents the set of buffers for all $k \in K$. The set B_I denotes the set of input buffers and the set B_O denotes the set of output buffers. The set $E_I \subseteq B_I \times K$ denotes the set of edge dependencies between each input buffer and kernel, $E_O \subseteq K \times B_O$ denotes the set of edge dependencies between each kernel and output buffer. The set $E \subseteq B_O \times B_I$ denotes the set of input output buffer dependencies across kernels in the DAG. Command queues are typically setup per device depending on which kernels are mapped to which devices.

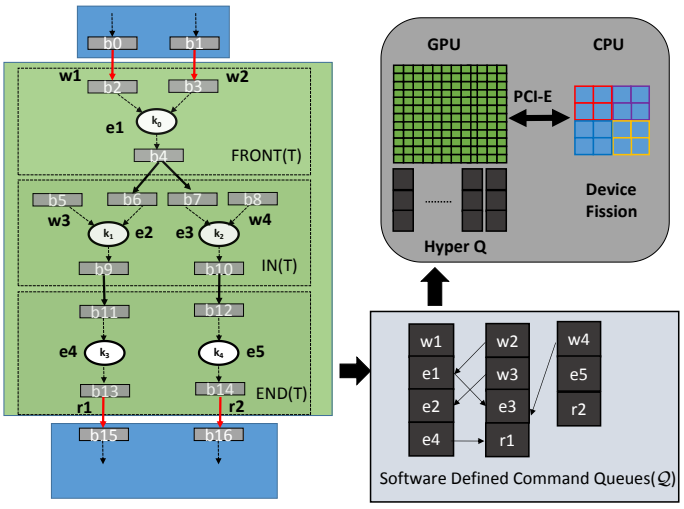

FIGURE 3.5: Platform and DAG Model

Given an OpenCL DAG $G = \langle K, B, E_I, E_O, E \rangle$, we denote a task component T_d as a subset of kernels $K' \subseteq K$ where each kernel k is mapped to the same device d. In our case, $d = \{cpu, gpu\}$. In Fig. 3.5, we have $T_{gpu} = \{k_0, k_1, k_2, k_3, k_4\}$. For a given task component we define the following terminology.

Definition 3.1. *Given a task component T_d pertaining to some OpenCL DAG G, we define $FRONT(T_d)$ as the set of kernels where each kernel k has input buffer dependencies $(b_i, k) \in E_I$ such that for b_i, if there exists an immediate predecessor b_j where $(b_j, b_i) \in E$ and $(k', b_j) \in E_O$, then the kernel k' belongs to a different task component $T'_{d'}$.*

In Fig. 3.5, we observe that $FRONT(T) = \{k_0\}$, since both input buffers $b2$ and $b3$ have predecessors pertaining to kernels belonging in a different task component.

Definition 3.2. *Given a task component T pertaining to some OpenCL DAG G, we define $END(T)$ as the set of kernels where each kernel k has output buffer dependencies $(k, b_i) \in E_O$ such that for b_i if there exists an immediate successor b_j where $(b_i, b_j) \in E$ and $(b_j, k') \in E_I$ then kernel k' belongs to a different task component $T'_{d'}$.*

In Fig. 3.5, we can observe that $END(T_{gpu}) = \{k_3, k_4\}$, since both output buffers $b13$ and $b14$ are used as inputs for kernels belonging to a different task component.

Definition 3.3. *Given a task component T_d pertaining to some OpenCL DAG G, we define $IN(T)$ as the set of kernels where each kernel $k \in T_d, k \notin FRONT(T), k \notin END(T)$.*

In Fig. 3.5, we can observe that $IN(T) = \{k_1, k_2\}$

We classify buffer edge dependencies $(b_i, b_j) \in E$ into two categories -i) intra-edge, ii) inter-edge.

Definition 3.4. *Given a task component T_d pertaining to a DAG G, an edge (b_i, b_j) is said to be intra-edge if there exists kernels k_i, k_j such that $(k_i, b_i) \in E_O$, $(b_j, k_j) \in E_I$ and both kernels k_i, k_j belong to the same component.*

In Fig. 3.5, we can observe that $(b4, b6)$ and $(b4, b7)$ are intra-edges for T_{gpu}.

Definition 3.5. *Given two task components T_x and T_y pertaining to a DAG G, an edge (b_i, b_j) is said to be an inter-edge from T_x to T_y if there exists kernels k_i, k_j such that $(k_i, b_i) \in E_O$, $(b_j, k_j) \in E_I$ and kernel k_i belongs to T_x and k_j belongs to T_y.*

In Fig. 3.5, we can observe that $(b0, b2)$, $(b1, b3)$, $(b13, b15)$ and $(b15, b16)$ are inter-edges. We classify kernel-buffer dependencies in E_I and E_O into two categories - i) isolated copy and ii) dependent copy

Definition 3.6. *Given any kernel k_i, an edge $(b_i, k_i) \in E_I$ represents an isolated copy (write)* iff *for every $b_k \in B$, $(b_k, b_i) \notin E$. In a similar fashion, an edge (k_i, b_j) represents an isolated copy(read)* iff *for every $b_k \in B$, $(b_j, b_k) \notin E$ respectively.*

In Fig. 3.5, the edges $(b5, k_1)$ and $(b8, k_2)$ correspond to isolated writes.

Definition 3.7. *Given any kernel k_i pertaining to a task component T_d, an edge $(b_i, k_i) \in E_I$ represents a dependent copy (write)* iff *there exists some buffer $b_i \in B$ such that $(b_i, b_k) \in E$. In a similar fashion, an edge (k_i, b_i) represents a dependent copy(read)* iff *there exists some $b_k \in B$ such that $(b_j, b_k) \in E$.*

In Fig. 3.5, every buffer-kernel dependency apart from $(b5, k_1)$ and $(b8, k_2)$ correspond to dependent copies.

Definition 3.8. *Given a task component T_d of an application DAG G mapped to a device d with r command queues, we define the command queue data structure $\mathcal{Q}$ as a graph $\langle V_Q, E_Q \rangle$ where $V_Q = \{q_1, q_2, \cdots, q_r\}$ denotes the set of command queues allocated to T_d. Each element o_i belonging to each queue q_i constitutes either a write, ndrange or read operation pertaining to some kernel belonging to T_d. Each element of the set E_Q is an edge of the form $\langle o_i, o_j \rangle$ where $o_i \in q_r$ and $o_j \in q_s$ such that $q_r \neq q_s$ and represents the precedence constraints enforced by the edges in the DAG G.*

The framework uses an *enq* procedure that sets up the data structure $\mathcal{Q}$ for task component T_d as follows. An operation o_i pertaining to kernel k_i is enqueued to one of the command queues in V_Q depending upon the membership of k_i in the sets $IN(T_d)$, $FRONT(T_d)$ and $END(T_d)$. This is described below.

i)If $k_i \in FRONT(T_d)$, the enqueue procedure *enq* enqueues all dependent write commands for buffers b_j corresponding to dependent writes $(b_j, k_i) \in E_I$ followed by the ndrange command for k_i.

ii) If $k_i \in END(T_d)$, the enqueue procedure *enq* enqueues the ndrange command for k_i followed by all dependent read commands for buffers b_j corresponding to intra-edges $k_i, b_j \in E_O$.

iii) If $k_i \in IN(T_d)$, the enqueue procedure *enq* enqueues only the ndrange command for k_i

iv) For every kernel k_i belonging to $FRONT(T)$, $IN(T)$and $END(T)$, the enqueue procedure *enq* enqueues all isolated writes for input buffers b_j, $(b_j, k_i) \in E_I$ before enqueuing the ndrange command for k_i and all isolated reads for output buffers b_r, $(k_i, b_r) \in E_O$ after enqueueing the ndrange command for k_i.

An edge $(o_i, o_j) \in E_Q$ exists if i) o_i is a isolated/dependent write (b_r, k_s) and o_j is an ndrange operation for kernel k_s ii) o_i is an ndrange operation for kernel k_s and o_s is a dependent/isolated write (k_s, b_r) and iii)both o_i and o_j are ndrange operations for kernels k_r and k_s respectively such that there exists edges $(k_r, b_r) \in E_O$, $(b_s, k_s) \in E_I$ and $(b_r, b_s) \in E$ is an intra edge. In Fig. 3.5, (b_3, k_0) corresponds to a dependent write for kernel k_0 thus requiring a dependency between associated operations w_2

and e_1 in $\mathcal{Q}$. The edge (e_1, e_3) represents the dependency between kernels k_0 and k_2 arising due to the dependencies (k_0, b_4),$(b4_b7)$,(b_7, k_2) where b_4, b_7 is an inter edge.

The framework expects that the device preferences for each kernel are known beforehand. Using this information and the *enq* procedure, the framework can emulate dynamic coarse-grained scheduling decisions supported by frameworks like StarPU and SOCL where kernels are dispatched one at a time to devices. For fine-grained scheduling algorithms it is expected that the user provides an initial decomposition of the DAG G into a set of task components $\mathcal{T}$ where each task component $T_d \in \mathcal{T} \in \mathcal{T}$ is mapped to a particular device d. Additionally, one must provide as guidance parameters for each task component T_d, the number of command queues to be used. Given this as input, the framework automatically sets up multiple command queues inside each task component for device d and outputs a schedule σ which is an ordered sequence of *enq* procedures that respects the precedence relationships of the application DAG. The problem formulation is formally stated as follows.

Definition 3.9. *Given a DAG* $G = \langle K, B, E_I, E_O, E \rangle$*, a corresponding set of* m *task components* $\mathcal{T} = \{T_{d_1}, T_{d_2}, \cdots, T_{d_p}\}$ *and a target heterogeneous CPU-GPU multicore platform* $\mathcal{P} = \{d_1, d_2, \cdots d_p\}$ *containing* p *devices, a schedule* σ *is an ordered sequence of enqueue procedures* $enq(T_{d_1}), enq(T_{d_2}), \cdots, enq(T_{d_p})$ *such that the kernels* $k_i \in K$ *is dispatched in a topologically sorted fashion i.e sorted with respect to the ordering of* k_i*'s enforced by the edges in* G*,* $k_1 \preceq k_2 \preceq \cdots, k_{|K|}$.

In the remaining sections of the paper, we discuss the intricate details of both the frontends and the backends of the framework.

Chapter 4

Software Architecture

An overview of the software architecture for *PySchedCL* is depicted in Fig. 4.1. The framework consists of two distinct modules, the functionalities of which are elaborated as follows.

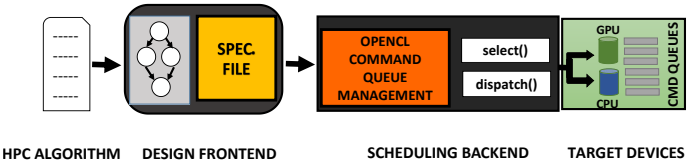

FIGURE 4.1: PySchedCL Toolflow

1. **Design Frontend:** The input to the scheduling framework is an OpenCL application represented in the form of a task graph G as discussed earlier. The proposed framework supports a specification file using which programmers can easily design an OpenCL application for execution on a heterogeneous platform. The programmer requires to implement only the OpenCL kernels and provide configuration parameters such as the dimensions of the input/output dataspace along with dependency information between kernels in this specification file.

2. **Scheduling Backend:** The scheduling backend takes as input the specification file in Step 1, and schedules the computation of each data parallel kernel in the application across the devices of a heterogeneous CPU/GPU platform.

This is achieved with the help of select and dispatch routines. The framework uses the `select` function to i)choose a task component from G and ii) select an available device (CPU/GPU). It next uses the `dispatch` (line 8) routine for finally issuing relevant write, ndrange and read commands now specified in $\mathcal{Q}$ to target devices. The backend API also supports end designers to implement custom scheduling heuristics by overriding the functionalities of the `select` and `dispatch` routines.

The modular design of the scheduling backend is an integral feature of the framework. The default `select` routine expects a static assignment of kernels to devices prior to the execution. It can be substituted by the user very easily with other scheduling policies which can even be dynamic in nature. The experimental section of this thesis extensively uses this modular feature of the `select` function to implement and compare several scheduling policies on the transformer DAG. Additionally, experiments with a Machine Learning based scheduling policy have also been performed in a similar fashion.

We next elicit implementation details for each of the two modules constituting the framework in the following subsections.

4.1 Input File Specification

The specification file used in *PySchedCL* is written using the Javascript Object Notation (JSON) file format. The file consists of a collection of key value pairs depicting necessary attribute information for an OpenCL kernel which includes information regarding input/output buffers, variables passed as arguments to the kernel call, the dimension of the kernel etc. Our tool processes this specification file and uses the scheduling backend to mimic the execution of a host program executing this kernel.

We have a LLVM compiler pass which parses the abstract syntax tree of an OpenCL kernel and generates an incomplete JSON file. The pass understands the dimensionality of the kernel, the types and positions of each variable and buffer used in the kernel function call. It also classifies buffers as input/output buffers by understanding whether it is treated as *l-values* or *r-values* in the body of the function. Given

this file, the user is only required to specify guidance parameters which include -i) the size of the buffers ii) the number of work items iii) the values of the variable arguments and iv) the device and the number of command queues to be used. The user has the option of either hardcoding constant numbers or writing expressions containing symbolic variables that depicts the relationship between work items and the dataspace to be processed. This ensures that we have one specification file for a kernel and the final values of these symbolic variables can be provided as command line parameters at runtime. We explain this by designing a JSON specification file for the matrix multiplication kernel depicted in Listing 1.

```
__kernel void gemm(__global float *A, __global float *B, __global float *C,
int M, int N, int K) {
int ty = get_global_id(1);
int tx = get_global_id(0);
if ((tx < N) && (ty < M))
{
C[ty * N + tx] = 0;
for(int k=0; k < K; k++)
C[ty * N + tx] += A[ty * K + k] * B[k * N +tx];
}

}
```

Listing 4.1: OpenCL Kernel for Matrix Multiplication

The matrix multiplication kernel is a 2-D kernel which takes as input two matrices A, B of dimensions $M \times K$, $K \times N$ respectively and produces an output matrix C of dimension $M \times N$. A total of $M * N$ work items is launched where the job of each work item is computing the dot product of one row of A and one column of B to produce one element of C. The JSON specification file for the same is depicted in Fig. 4.2

The file comprises the following information.

1. **Kernel Information:** This includes i) the name of the OpenCL kernel function (which is `gemm` in Fig. 4.2), ii) the filepath of the required kernel source file, iii) the dimensionality of the kernel in `workDimension` and iv) the total number of work items (`globalWorkSize`) to be launched for this kernel. The variable `globalWorkSize` is a three element list where each element refers to the number of work items along a particular dimension. As guidance parameters, the user can specify these elements either as compile time constants or using an generic expression containing symbolic variables. For the example

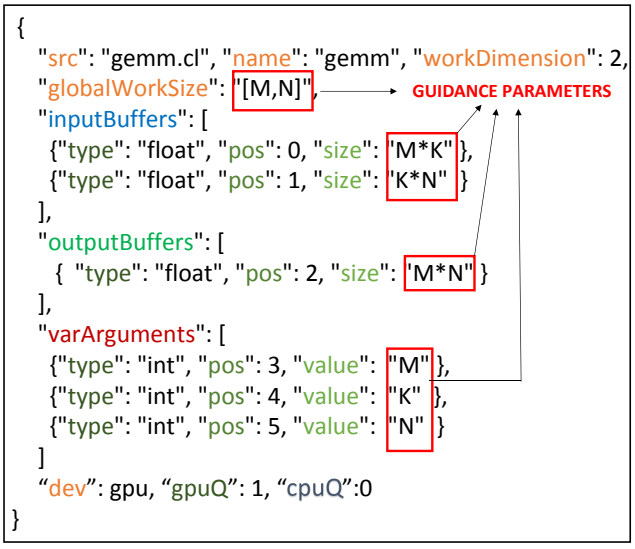

FIGURE 4.2: JSON File for Matrix Multiplication

JSON file, we have `globalWorkSize = [M,N,1]`. The values of M and N can be configured as command line parameters right before dispatching the kernel.

2. **Buffer Information:** The JSON file maintains information for three buffer lists - `inputBuffers` reserved for input buffers, `outputBuffers` reserved for output buffers and `ioBuffers` reserved for buffers which are treated as both input and output by the kernel. Each buffer belonging to any one of the lists is characterized by the tuple $\langle type, size, pos \rangle$ where *type* denotes the data type for each element in the buffer, *size* denotes the total number of elements in the buffer, and *pos* denotes the index position of the buffer argument in the actual function call of the kernel. For example, the input buffer passed as argument in the first position of the function call in Listing 1 has $pos = 0$. The user can configure the guidance parameter *size* for each buffer either as a compile time constant or an expression of symbolic variables. For the example JSON file in Fig. 4.2, the sizes of the buffers are the number of elements for each matrix.

3. **Kernel Arguments:** In the JSON file, each variable argument passed as an argument to the OpenCL function call is denoted by the tuple $\langle type, value, pos \rangle$ where *type* denotes the type of the variable, *value* represents the value contained in the variable and *pos* represents the index position of the variable argument in the actual function call of the kernel. The user can configure

the guidance parameter *value* again either as a compile time constant or as a symbolic variable. In Fig. 4.2, we have three variable arguments M, N, K each depicting the size of one dimension of the matrices.

4. **Device Information:** Finally, the *dev* field indicates which device to be used. The fields *gpuQ* and *cpuQ* denote the number of command queues to be used for the GPU and the CPU devices respectively.

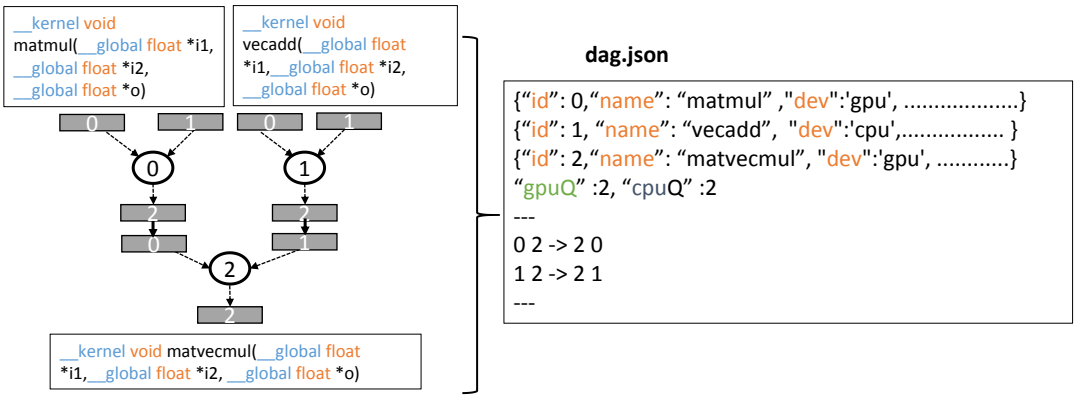

FIGURE 4.3: OpenCL DAG Specification

In general a DAG of OpenCL kernels, can be specified as a single JSON file which contains the information regarding each kernel generated by running the compiler pass on each kernel. The user additionally has to specify information that captures the precedence constraints of the DAG. Unlike frameworks like SOCL, StarPU and MultiCL which requires implementing host-side implementations, our framework relies only on the DAG specification provided as a simple JSON file with configuration parameters of individual kernels and dependency information of the DAG.

Let us consider an example DAG comprising three kernels as depicted in Fig. 4.3. Each kernel now is designated by a unique identifier field called *id*. The kernel with id 0 represents matrix multiplication kernel which takes as input two matrices of dimensions $M \times K$ and $K \times N$ and produces an output of dimension $M \times N$. The corresponding sizes of the input and output buffers are specified inside the JSON file of the kernel using these symbolic variables. The kernel with id 1 represents a vector addition kernel which takes two vectors of size N and produces one output vector of size N. The kernel with id 2 represents matrix-matrix multiplication kernel which takes as input an $M \times N$ matrix and $N \times 1$ vector and produces an $M \times 1$

vector. Again, the corresponding sizes of the input and output buffers for these kernels are specified in the individual JSON files of the kernels. The outputs of the kernels 1 and 2 are used by the kernel with id 2. The dependency information for the same is specified as a set of edges of the form $k_i, b_r \rightarrow k_j, b_s$, where k_i,k_j represent kernel ids that are dependent, b_r is an output buffer of k_i and b_s is an input buffer of k_j i.e. $(k_i, b_r) \in E_O$, $(b_s, k_j) \in E_I$ and $b_r, b_s \in E$. The ids for the buffers b_r and b_s are represented by their corresponding argument positions in the function call for the kernels. For example, if we consider $0, 2 \rightarrow 2, 0$, the output buffer specified in argument 2 of kernel 0 will be used as input buffer specified in argument 0 of kernel 2. As guidance parameters, the user can specify the device preference for each kernel using the `dev` field. In Fig. 4.2, the kernels with ids 0 and 2 are mapped as a task component to the GPU device while kernel with id 1 is mapped to the CPU device. The framework automatically takes care of setting up command queues specified in the fields *cpuQ* and *gpuQ* so as to maximally exploit concurrency using the scheduling engine which is discussed next.

4.2 Scheduling Backend

An overview of the scheduling backend is depicted in Fig. 4.4. We explain the working principle of the backend with the help of the procedure *schedule* highlighted in Algorithm **??** for scheduling. The input to *schedule* is the application graph G and the set of devices in the target platform $\mathcal{P}$. The procedure first parses the input specification and extracts the set of task components that are ready for dispatch (line 2). We say a task component T is ready for dispatch if for every kernel $k_i \in FRONT(T)$, i) there exists no predecessor or ii) all predecessors of k_i have finished execution. Each such task component is pushed into the centralized task queue $\mathcal{F}$ for dispatch. We note that task components can be individual kernels or a collection of kernels as specified by the guidance parameters in the specification file. For the former case, the decisions are coarse-grained in nature i.e. all the operations associated with a kernel k_i are finished before proceeding to execute a successor kernel k_j. The set $\mathcal{A}$ represents the set of available devices and is initialized to all the devices contained in $\mathcal{P}$. The procedure *schedule* operates on tasks belonging to the task queue $\mathcal{F}$ by selecting and dispatching free task components whenever

there exists an available device in $\mathcal{A}$ (lines 4-8). This is continued until all kernels of the DAG have been processed (line 3). The *select* routine is used to select a task component T and an empty command queue data structure $\mathcal{Q}$ for an available device belonging to $\mathcal{A}$. The number of command queues to be used in $\mathcal{Q}_d$ for a device d is specified as a guidance parameter by the user in the specification file.

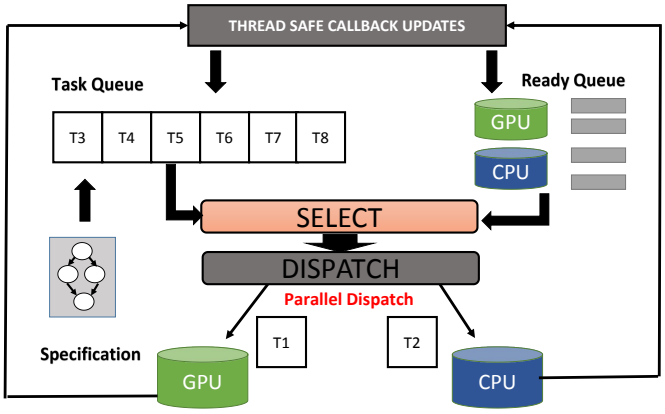

FIGURE 4.4: Command Queue Setup

Appendix A

Appendix A

Write your Appendix content here.

Bibliography

[1] Aji, A. M., Peña, A. J., Balaji, P., and chun Feng, W. (2016). Multicl: Enabling automatic scheduling for task-parallel workloads in opencl. *Parallel Computing*, 58:37–55.

[2] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In Sips, H., Epema, D., and Lin, H.-X., editors, *Euro-Par 2009 Parallel Processing*, pages 863–874, Berlin, Heidelberg. Springer Berlin Heidelberg.

[3] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., and Hwu, W.-m. W. (2010). An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, New York, NY, USA. ACM.

[4] Ghose, A., Dokara, L., Dey, S., and Mitra, P. (2017). A framework for opencl task scheduling on heterogeneous multicores. *PPL*, 27(3-4):1–32.

[5] Grewe, D. and O'Boyle, M. F. (2011). A static task partitioning approach for heterogeneous systems using opencl. In *CC*, pages 286–305. Springer.

[6] Grewe, D., Wang, Z., and O'Boyle, M. F. (2013). Opencl task partitioning in the presence of gpu contention. In *LCPC*, pages 87–101. Springer.

[7] Han, T. D. and Abdelrahman, T. S. (2011). hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90.

[8] Henry, S., Denis, A., Barthou, D., Counilh, M.-C., and Namyst, R. (2014). Toward opencl automatic multi-device support. In Silva, F., Dutra, I., and Santos Costa, V., editors, *Euro-Par 2014 Parallel Processing*, pages 776–787, Cham. Springer International Publishing.

[9] Hoshino, T., Maruyama, N., Matsuoka, S., and Takaki, R. (2013). Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 136–143.

[10] Hugo, A., Guermouche, A., Wacrenier, P., and Namyst, R. (2013). Composing multiple starpu applications over heterogeneous machines: A supervised approach. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1050–1059.

[11] Jääskeläinen, P., Korhonen, V., Koskela, M., Takala, J., Egiazarian, K., Danielyan, A., Cruz, C., James, P., and McIntosh-Smith, S. (2018). Exploiting task parallelism with opencl: A case study. *Journal of Signal Processing Systems*. EXT="Danielyan, Aram".

[12] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. (2012). Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 341–352, New York, NY, USA. Association for Computing Machinery.

[13] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A. (2012). Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 38(3):157–174.

[14] Kofler, K., Grasso, I., Cosenza, B., and Fahringer, T. (2013). An automatic input-sensitive approach for heterogeneous task partitioning. In *SC*, pages 149–160. ACM.

[15] Nvidia (2010). Nvidia gpu computing sdk.

[16] Steuwer, M., Kegel, P., and Gorlatch, S. (2011). Skelcl - a portable skeleton library for high-level gpu programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182.

[17] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66.

[18] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

[19] Wen, Y., Wang, Z., and O'Boyle, M. F. P. (2014). Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *HiPC*, pages 1–10.

[20] Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J., and Feng, W. (2012). Vocl: An optimized environment for transparent virtualization of graphics processing units. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12.

[21] You, Y.-P., Wu, H.-J., Tsai, Y.-N., and Chao, Y.-T. (2015). VirtCL: a framework for OpenCL device abstraction and management. In *Proceedings of the 20th Symposium on Principles and Practice of Parallel Programming*, pages 161–172. ACM.