PREDICTABLE GPGPU COMPUTING IN DNN-DRIVEN AUTONOMOUS SYSTEMS

by

Husheng Zhou

APPROVED BY SUPERVISORY COMMITTEE:

Cong Liu, Chair

Farokh B. Bastani

András Faragó

Lingming Zhang

To my family.

PREDICTABLE GPGPU COMPUTING IN DNN-DRIVEN AUTONOMOUS SYSTEMS

by

HUSHENG ZHOU, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2018

ACKNOWLEDGMENTS

First and foremost, I am deeply grateful to my advisor, Professor Cong Liu, for his incredible guidance, endless support and encouragement throughout my entire PhD life. He introduced me to the field of real-time systems. He taught me from scratch on how to do research from an idea to problem solving and presenting a paper. His passion in research always inspires me to be a better researcher.

I also would like to express my gratitude to all of my collaborators, Zheng Dong, Soroush Bateni, Yuchuan Liu, Xia Zhang, Yue Ma, Yuankun Zhu, and Mozi Chen. I am deeply indebted for their tremendous help, both in work and life. I cherish my time working with them. I am also grateful to my mentors of internship, Dr. Liana Fong and Dr. Wei Tan, for giving invaluable guidance during my summer intern at IBM T. J. Watson Research Center.

My life in Dallas would not be as easy and happy without my friends. I would like to particularly thank a number of my friends: Yufei Gu, Yangchun Fu, Junyuan Zeng, and friends in Plano Chinese Alliance Church. Friendships with them made all the difference in the past six years.

Finally, my family members have always been there for me with unconditional love and support. This dissertation is dedicated to my parents, my awesome wife, Ruohan Zhang, and my cute baby girl, Chloe Zhou.

October 2018

PREDICTABLE GPGPU COMPUTING IN DNN-DRIVEN AUTONOMOUS SYSTEMS

Husheng Zhou, PhD
The University of Texas at Dallas, 2018

Supervising Professor: Cong Liu

Graphics processing units (GPUs) are being widely used as co-processors in many domains to accelerate general-purpose workloads that are data-parallel and computationally intensive, i.e., GPGPU. An emerging usage domain is adopting GPGPU to accelerate inherently computation-intensive Deep Neural Network (DNN) workloads in autonomous systems. Such autonomous systems are usually time-sensitive, especially for autonomous driving systems. When driving alongside human drivers, loss of life or property may result if the computing systems of the autonomous vehicles fail to respond to events before its deadline. Much research has been conducted to algorithmically optimize the accuracy and performance of deep neural networks, but limited attention has been given to optimizing the execution of GPU-accelerated DNN workloads from the scheduling angle, especially in a time-constrained multi-tasking environment.

Adopting GPGPU to accelerate DNN workloads in time-sensitive autonomous systems that are often resource-constrained presents a series of challenges: (1) GPUs are designed to execute non-preemptively, which may cause priority inversion; (2) How to optimize the execution of GPU-accelerated DNN workloads at the system level in a real-time multi-tasking environment; (3) How to simultaneously achieve two (often) conflicting goals in a resource-constrained embedded CPU-GPU heterogeneous platform: timing predictability and energy efficiency, that are essential for any DNN-based autonomous driving system.

The goal of the research presented in this dissertation is to solve or remedy the aforementioned challenges. Specifically, we propose GPES, a runtime system that allows GPU executions to be interruptible and preemptable in a multi-tasking environment. We proposed S^3DNN, a systemic solution that optimizes the execution of DNN workloads on GPU in a soft real-time multi-tasking environment. We proposed PredJoule, a runtime system which presents a layer-based approach that controls the timing and optimizes energy efficiency by exploiting each layer's performance/energy characteristics. In addition to the runtime systems we proposed, we investigate the problem of mapping multiple applications implemented using kernel graphs in a heterogeneous system, and present a theoretical framework that formulates this problem as an integer program and a set of practically efficient mapping algorithms. Furthermore we present a reuse-based approach to further improve the predictability of GPU computing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Graphics processing units (GPUs) are being widely used as co-processors in many domains to achieve acceleration. They are particularly capable of executing data-parallel applications, due to their highly multi-threaded architecture and high-bandwidth memory. Along with the support of the CUDA (NVIDIA, 2011) programming model developed by NVIDIA, GPUs can be easily used for general-purpose computing in addition to dedicated graphics applications, i.e., GPGPU. Examples include adopting GPGPU to accelerate inherently computation-intensive Deep Neural Network (DNN) workloads.

Deep Neural Network is another very popular technique being widely applied in many autonomous systems for their state-of-the-art, even human-competitive accuracy in cognitive computing. One such domain is autonomous driving, where DNNs are used to map the raw pixels from on-vehicle cameras to the steering control decisions (Chen et al., 2015; NVIDIA, 2016). This DNN-driven approach is powerful because with limited training data from humans, the driving system can learn to drive by itself. Recent end-to-end learning frameworks make it even possible for DNNs to learn to self-steer from limited human driving datasets (Bojarski et al., 2016).

Such autonomous systems are usually time-sensitive, sometimes even need to meet hard real-time constraints (no task should violate its deadline, otherwise the entire system would fail). However, the GPGPU is mainly designed for accelerating particular high-performance applications, which may not be efficiently applicable for GPGPU in real-time multi-tasking environments. Once pieces of GPU-accelerated code, i.e., kernels, from different applications are loaded onto the GPU, they are dispatched by hardware scheduler. Such hardware-based scheduling will harm the response time of high-priority GPGPU tasks, since the hardware scheduler does not consider task priorities. Consequently, due to the asynchronous and non-preemptive nature of GPU processing, in multi-tasking environments, a task with higher priority or urgency (e.g., with a shorter deadline) may be blocked by lower priority tasks that have already started running on GPUs. This severely

harms the system's timing predictability and is a serious impediment limiting the applicability of GPGPU in autonomous systems.

In addition to preemption, the steps that map computations to different processing elements in a CPU-GPU heterogeneous platform is critical. This mapping problem is quite challenging due to a large size of the policy space. First of all, applications may demonstrate (sometimes significantly) different performance characteristics when executed on GPUs than CPUs. The mapping algorithm thus needs to consider such heterogeneity when making prioritization and mapping decisions. Moreover, most real world workloads are implemented using rather complex kernel graphs, where a kernel graph contains a number of data- or logical- dependent kernels. The precedence constraints among kernels require the mapping algorithm to consider: *(i)* the kernel graph structure and *(ii)* different data transfer costs among kernels if executed on different processors. Furthermore, for data-intensive kernels, data partitioning techniques need to be incorporated into the mapping algorithm because partitioning a kernel into threads that can be run on multiple devices in parallel improves the overall utilization.

Furthermore, for DNN specific workloads, existing research works focus on exploring the specific features of DNN to improve the single-tasking throughput at the algorithmic level (Redmon et al., 2016; Girshick et al., 2014; Girshick, 2015; Ren et al., 2015). There is a lack of research effort tackling these challenges from the critical system-level optimization perspective: how to optimize the execution of GPU-accelerated DNN workloads at the system level in a real-time multi-tasking environment. A critical objective is to guarantee real-time performance while maximizing system throughput and resource utilization to mitigate the inherent resource constraint imposed by most embedded hardware.

Last but not least, adopting inherently compute-intensive DNNs in often resource- and energy-constrained automobiles creates another challenge, due to the need of satisfying two (often) conflicting goals: timing predictability and energy efficiency. Timing predictability (i.e., meeting job latency requirement) is one of the most important tenets in certification required for autonomous

driving systems. The functional correctness of an automobile hinges crucially upon temporal correctness (e.g., performing object detection within a strict latency boundary to signal automatic brake requests). On the other hand, automobiles demand low energy consumption, due to their strict size, weight, and power (SWaP) requirements. Regrettably, timing predictability and energy efficiency are often in conflict. This is because the former requires reserving sufficient resources for guaranteeing latency even in the worst case; while the latter often desires allocating just enough resource that barely meets the needs of the current job.

This dissertation seeks to investigate aforementioned challenges of adopting GPU in DNN-driven atonomous systems, aiming at making GPGPU computings timing- and energy- predictable.

1.1 Graphics Processing Units

GPUs were developed for dedicated 2D graphics rendering since 1970s. For two decades of years GPUs were "fixed function" hardware. This changed when the "programmable pipeline" appeared in 2001, which enabled programmers to customize their own rendering codes (namely "shaders") that were executed on the GPU. These successful shader languages include NVIDIAs "C for Graphics" (Cg) (NVIDIA, 2003), and the OpenGL Shading Language (GLSL) (Group, 2004). Empowered by shader languages and programmable GPUs, researchers began to exploit the generality of the programmable pipeline to solve general purpose computations, namely GPGPU (Harris, 2009). Recognizing the potential computational power of GPGPU, generalized languages and easy-to-use runtime environments were developed by major GPU vendors and software producers to allow general purpose programs to be executed on graphics hardware. Notable toolkits include the Compute Unified Device Architecture (NVIDIA, 2011), OpenCL (Group, 2008), and OpenACC (OpenACC, 2013).

Figure 1.1 shows the historical trends of CPU and GPU performance, where the x-axis is the time till August 2014 when this figure is used in the presentation of (Zhou et al., 2015), and the y-axis is the single precision peak performance in terms of billions of floating point operations

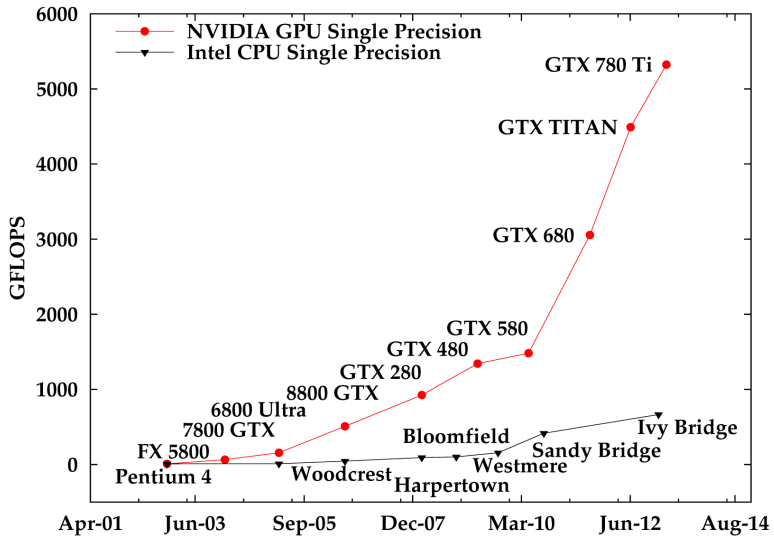


Figure 1.1: Historical trends of CPU and GPU performance in GFLOPS.

per second (GFLOPS). We observe that GPUs significantly outperform CPUs since June 2003, and this disparity becomes even larger. For example, NVIDIA's GTX Titan can perform at 4,500 GFLOPS in comparison to 672 GFLOPS for the Intel Ivy Bridge, which is more than 6 times. This performance trend shows the strong motivation of using GPUs to accelerate general purpose computing.

We briefly explain some GPU hardware related terminologies that will be frequently used in the rest of this dissertation.

MP/SM, SP: Streaming multiprocessor (MP or SM) is the internal unit of NVIDIA GPU hardware that performs the actual computations. NVIDIA GPU consists of several SMs, each of which is further divided into shader processors (SP). The number of SMs and SPs is product-specific. Low-end GPUs typically have one SM, and high-end GPUs have 15 or 16 SMs. Take NVIDIA GTX 480 for example, which is based on GF110 architecture, it has 15 SMs.

Channel: Each GPU context is associated with a GPU hardware channel. Internally, a channel is managed by channel engine which is a subarea of the MMIO (memory-mapped I/O) region. The channel engine maintains the status of GPU contexts including FIFO queues of GPU commands.

Command: Typically the GPU is controlled by the CPU using commands. The operating system and GPU driver maintain GPU *command buffers* which are accessible to both CPU and GPU. The CPU writes commands to them, while the GPU reads the commands from them. There are hundreds of commands defined by the architecture (e.g., Fermi or Kepler). For example, when copy data from the host to the device memory, we send a set of commands to the GPU, specifying the source and the destination virtual addresses together with the mode of direct memory access (DMA). A single GPU command is composed of GPU instructions and the values passed to the instructions. It represents atomicity operation. Commands are usually grouped as non-preemptive regions called *command group*. A tuple of size and command group address forms the packet written to command buffer.

GPU page table To enable multi-tasking on GPU, newer versions of GPUs since Fermi architecture (NVIDIA, 2010) support virtual addressing by assigning an address space to each application. An address space is defined by a GPU page table containing entries of mappings between virtual addresses and physical addresses. GPU kernels operate on virtual addresses which are transparently translated to physical addresses by a dedicated memory management unit (MMU). GPU page table is located in the PCI configurable space, which is accessible from both CPUs and GPUs.

GPU Memory Hierarchy To improve system throughput, NVIDIA GPUs typically feature several memory spaces and memory types. According to CUDA specifications, there are eight logical types of GPU memory and cache space.

Global memory GPU global memory is the largest area visible to all threads within the application (including the host), which is used to store input and output bulky data for GPU computations. The spatial isolation of global memory is managed by NVIDA proprietary driver which prevents a global memory object that belongs to one context from being accessed by another context.

Shared memory (or SMEM) GPU shared memory is a small yet fast memory type that resides on each SM. It can be directly operated by the GPU kernel code. It is visible to all threads within the same thread block. Its life duration is the same as the currently executing thread block while it is

flushed by GPU hardware only when the current context is destructed. It does not have any spatial isolation, which implies that data residuals on the SMEM can be accessed by other thread blocks within the same kernel or from different kernels.

L1 cache For each SM, there is a high speed cache named L1 cache, which is used for global memory load caching. It cannot be directly operated by GPU kernel code. However, in some GPU architectures (e.g., Fermi, Kepler), L1 cache and shared memory physically share the same hardware resources, making L1 cache readable by crafted GPU kernel code.

Local Memory GPU local memory is not a physical type of memory, but an abstraction of global memory, which is only used to hold automatic variables. Its scope is local to the thread and it resides off-chip.

Register GPU registers represent the fastest yet the smallest memory on GPU. As a rare resource, register pressure may severely detract performance. Register pressure occurs when there are not enough registers available for a given kernel. When this occurs, the data is "spilled over" using local memory.

1.2 GPGPU Programming Model

GPGPU applications typically obey the following execution flow: (i) initializing the GPU device, (ii) allocating GPU device memory, (iii) transferring data from host memory to device memory, (iv) launching the computation work (kernel) on GPU, (v) copying results back to host memory, and (vi) freeing device memory and closing the device. We highlight some CUDA specific conceptual terminologies that will be frequently used in the rest of this dissertation.

Context: Context conceptually represents separate virtual address spaces on the GPU hardware. A context is either transparently or explicitly created for a CUDA application at the GPU device initialization stage. NVIDIA has provided the MPS (Multi-Process Service) feature in newer versions of CUDA to transparently merge multi-process CUDA applications into one context. However, the usage of MPS is limited by operating system (only supports Linux-based system), applications

(only supports 64-bit applications), GPU hardware (compute capability 3.5 or higher), and special configuration (set GPU to be exclusive to other processes). Moreover, MPS fails to consider DNN-specific characteristics since it is an application-oblivious approach and will blindly combine all of the processes assigned to it. In this dissertation, we assume a more common scenario, where different CUDA applications run on different contexts.

Thread, Warp, Block, Grid: The NVIDIA CUDA (NVIDIA, 2011) programming model consists of four levels of hierarchy. In Fermi or Kepler (NVIDIA, 2014) architecture, 32 threads make up a warp. Warps are the basic units of execution on the GPU. Threads in each warp are executed together. A group of warps stitch together to form a block. These blocks are combined to form a grid. A grid is corresponding to an execution kernel, thus in the rest of this dissertation, kernel launch and grid launch are interchangeable forms. When executing a kernel, the corresponding entire grid is mapped to one GPU device, blocks are mapped to SMs (MPs), and internally, computations are scheduled warp by warp. Notice that, in Fermi and Kepler architectures, grids from different kernels can execute on the same GPU device simultaneously, which is so called concurrent kernels. But such grids (kernels) must come from the same context. In CUDA programming, the programmer can control the number of threads within a block and the number of blocks within a grid.

CUDA stream: CUDA stream is a technique introduced by a newer version of CUDA (NVIDIA, 2015), which aims to hide the latency of memory copy and kernel launch from different independent operations. Its additional effect is the capability of enabling concurrent kernel execution which allows multiple kernels to execute on the same GPU simultaneously when each kernel cannot fully utilize the entire GPU device. A CUDA stream can be explicitly created by the programmer and bound to a kernel launch or data copy operation. To avoid confusion with data stream/video stream, when we talk about this technique, we use the term CUDA stream.

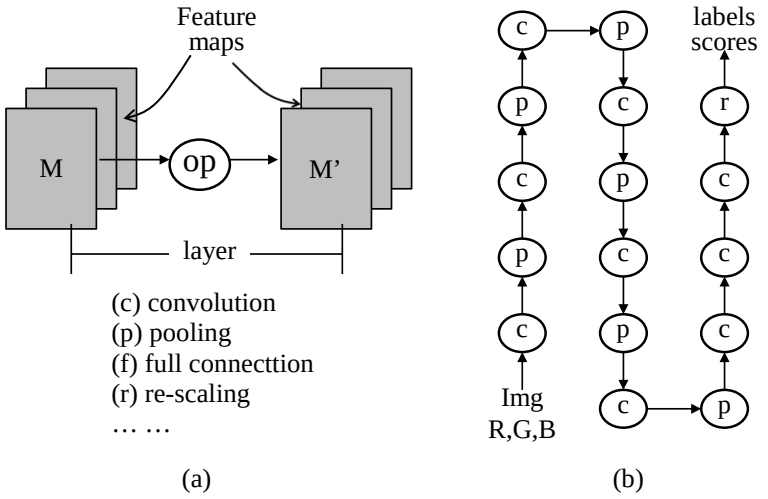


Figure 1.2: (a) DNN layers are essentially array-based computations operated on lists of arrays often called feature maps. (b) A state-of-the-art DNN for object recognition, formed by connected layers.

1.3 Deep Neural Networks

A DNN can be viewed as a dataflow graph, in which its nodes, or layers, are essentially array-based computations (as shown in Fig. 1.2(a)) (Redmon et al., 2016). Each layer takes a set of arrays, called ***feature maps*** as input and outputs a set of feature maps that will in turn be processed by subsequent layers belonging to the same DNN instance. Fig. 1.2(b) shows an illustration of a set of DNN layers used in YOLO (Redmon et al., 2016), which is a popular DNN framework aimed at real-time object detection. The letter within each cycle denotes the functionality performed by the corresponding layer: 'c' for convolution layer which convolves inputs by a convolution filter, 'p' for a max pooling layer which replaces each input array value by the maximum of its neighbors, 'f' for a fully connected layer which multiplies the feature maps by a weight matrix, and 'r' for a region layer, which is a layer specific to YOLO that is responsible for finding areas of interest in an image. In order to use a DNN for object detection, a pre-trained weight file is needed which is estimated from training data beforehand. Training an accurate weight file is an offline procedure that usually takes several days or weeks, done by "learning" features from large-scale

image datasets. In this dissertation, we are not concerned with improving the training process, rather, we focus on efficient execution of DNNs for real-time object detection.

DNN in Autonomous Driving. An autonomous driving system captures surrounding environmental data via multiple sensors (e.g., camera, Radar, Lidar) as inputs, processes these data with DNNs and outputs control decisions (e.g., steering). In this dissertation, we mainly focus on the steering angle component with camera inputs and steering angle outputs, as adopted in NVIDIA Dave (Bojarski et al., 2016). Convolutional Neural Network (CNN), which is efficient at analyzing visual imagery, is the most widely used DNN for steering angle decisions. Similar to regular neural networks, CNNs are composed of multiple layers and pass information through layers in a feed-forward way. Among all layers, the convolutional layer is a key component in CNNs, which performs convolution with kernels on the output of previous layers and sends the feature maps to successor layers. Different from another widely used DNN architecture – Recurrent Neural Networks (RNNs) which is a kind of neural network with feedback connections, CNN-based steering model makes steering decisions based only on the currently captured image. In the dissertation, we use DNN to represent both CNN and RNN without explicitly differentiating them.

1.4 Contributions

We now present an overview of the contributions of this dissertation that support this thesis.

- **Making GPU execution (partially) preemptive.** We present an efficient GPGPU preemptive execution system (GPES), which combines user-level and driver-level runtime engines to reduce the pending time of high-priority GPGPU tasks that may be blocked by long-freezing low-priority competing workloads. GPES automatically slices a long-running kernel execution into multiple subkernel launches and splits data transaction into multiple chunks at user-level, then inserts preemption points between subkernel launches and memory-copy operations at driver-level. We implement a prototype of GPES, and use real-world benchmarks

and case studies for evaluation. Experimental results demonstrate that GPES is able to reduce the pending time of high-priority tasks in a multi-tasking environment by up to 90% over the existing GPU driver solutions, while introducing small overheads.

- **Improving real-time performance of DNN workloads in real-time multi-tasking environment.** We propose S^3DNN, a system solution that optimizes the execution of DNN workloads on GPU in a real-time multi-tasking environment, which simultaneously optimizes the two (sometimes) conflicting goals of real-time correctness and throughput. S^3DNN contains a governor that selectively gathers system-wide DNN requests to perform smart data fusion, and a novel supervised streaming and scheduling framework that combines a deadline-aware scheduler with the concurrency-enabled CUDA stream technique. To simultaneously maximize concurrency-induced benefits and real-time performance, S^3DNN explores a rather interesting and unique characteristic of DNN workloads, where multiple layers of a DNN instance often exhibit a gradually decreased GPU resource utilization pattern. We have fully implemented S^3DNN in a GPU-accelerated system and have conducted extensive sets of experiments evaluating the efficacy of S^3DNN under a wide range of system and workload scenarios. The results show that S^3DNN significantly improves upon state-of-the-art GPU-accelerated DNN processing frameworks, e.g., up to 37% and over 40% improvements in real-time performance and throughput, respectively.

- **Optimizing energy efficiency of DNN workloads while meeting timeliness.** We propose PredJoule, a timing-predictable energy optimization framework for running DNN workloads in a GPU-enabled automotive system. PredJoule achieves both latency guarantees and energy efficiency through a layer-aware design that explores specific performance and energy characteristics of different layers within the same neural network. We implement and evaluate PredJoule on the automotive-specific NVIDIA Jetson TX2 platform for five state-of-the-art DNN models with both high and low variance latency requirements. Experiments show

that PredJoule rarely violates job deadlines, and can improve energy by 65% on average compared to five existing approaches and 68% compared to an energy-oriented approach.

- **Improving throughput of multiple graph-struct applications by novel heterogeneous task mapping.** We investigate the problem of computation and data mapping for multiple applications while minimizing the completion time in GPU-CPU heterogeneous systems, by presenting a theoretical framework that yields an optimal integer programming solution. Moreover, based upon several interesting measurements-based case studies, we design three practical mapping algorithms with low time complexity, each of which explores a specific set of factors that may affect the completion time performance. We evaluated the proposed algorithms by implementing them on a real heterogeneous system and using a large set of popular benchmarks for evaluation. Experimental results demonstrate that our proposed algorithms can achieve up to 30% faster completion time compared to the state-of-the-art mapping techniques, and can perform consistently well across different workloads.

- **Improving throughput of clusters by reusing GPU computation results.** We present GRU, an ecosystem that smartly manages and shares GPU resources through exploiting redundancy. GRU transparently interprets GPU-accelerated computing requests and memoizes results for potential future reuse. To enhance reusability, GRU implements a partial result reuse idea, where GPU computation requests even with different input data and functionality may become reusable with respect to each other. To guarantee correctness of partial reuse, GRU employs a compiler-assisted approach that analyzes general data parallel patterns that are reliable for the reuse purpose, and is capable of smartly recognizing such reusable data parallel patterns of incoming requests. We have fully implemented GRU and conducted extensive sets of experiments running micro-benchmarks on local machines and real-world applications including Spark-based uses cases in an AWS cluster. Evaluation results show that GRU is effective in identifying and eliminating redundant GPU computations, achieving up to 5x (2.5x) speedup for compute-intensive (data-intensive) benchmarks. In addition,

GPU-managed Spark observes a reduction of 25.3% (39.8%) on average with respect to turnaround time (GPU occupation time) over state-of-the-art solutions.

1.5 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we discuss several background topics, and prior work on predictable GPGPU scheduling in autonomous systems. In Chapter 3, we describe the design, implementation, and evaluation of our preemptive GPU execution framework, GPES. In Chapter 4, we present S^3DNN– a systemic solution that optimizes the execution of DNN workloads on GPU in a real-time multi-tasking environment. In Chapter 5, we presents PredJoule, a timing-predictable energy optimization framework for running DNN workloads in a GPU-enabled automotive system, which presents a layer-based approach that controls the timing and optimizes energy efficiency through exploiting each layer's performance/energy characteristics. In Chapter 6, we investigate the problem of mapping multiple applications implemented using kernel graphs in a heterogeneous system consisting of CPUs and GPUs, in order to achieve fast competition time. In Chapter 7, we present GRU, a GPU sharing, result memoization and reuse ecosystem for high performance and cloud computing, in order to further improve the predictability of GPU computing. We end in Chapter 8 with concluding remarks and a discussion of future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Scheduling Algorithms for Heterogeneous Systems

The general problem of scheduling in heterogeneous systems has received much attention. A number of scheduling heuristics have been proposed for scheduling directed acyclic graph-based (DAG) applications in heterogeneous systems (Topcuouglu et al., 2002; Bittencourt et al., 2010; Zhao and Sakellariou, 2003; Arabnejad and Barbosa, 2014; Sakellariou and Zhao, 2004; Canon et al., 2008). These algorithms schedule a single DAG (Directed Acyclic Graph) of tasks onto heterogeneous processing units with varying speed for minimizing the completion time. Zhao et.(Zhao and Sakellariou, 2006) proposed multi-DAG scheduling by merging multiple DAGs into one DAG. However, such algorithms do not specifically target the CPU/GPU platform, and thus ignore several critical factors when making scheduling decisions, including non-preemptivity, data transfer cost among CPUs and GPUs, data partitioning. Moreover, these existing algorithms are mostly greedy in nature and do not provide a theoretical understanding of the mapping problem considered herein. Furthermore, such algorithms use simulation-based evaluation approach and have not been tested in real systems.

2.2 Runtime Engines for Heterogeneous CPU/GPU Processors

For heterogeneous CPU/GPU platforms, a number of runtime systems have been developed to perform task scheduling. PTask (Rossbach et al., 2011) focuses on eliminating performance interference of GPU sharing. TimeGraph (Kato et al., 2011) and others (Verner et al., 2011) provides prioritization and isolation capabilities in GPU resource management. Harmony (Diamos and Yalamanchili, 2008) schedules translated CUDA code on various devices. Qilin (Luk et al., 2009) provides an adaptive mapping to automatically partition tasks on a CPU and a GPU. SKMD (Lee

et al., 2013) transparently translates single OpenCL (Group, 2008) kernel into variations and executes them on multiple GPUs simultaneously. The aforementionned runtime systems either focus on single kernel or do not consider kernel affinities. Some other runtime systems focus on task dataflow parallelism: OmpSs (Bueno et al., 2011), DirectShow (Linetsky, 2001), Hydra (Weinsberg et al., 2008), StreamIt (Thies et al., 2002), IDEA (Currey et al., 2013), Liquid Metal (Huang et al., 2008), Lime (Auerbach et al., 2010). However, these systems do not focus on scheduling multiple graphs onto heterogeneous processors for minimizing the completion time.

The StarPU (Augonnet, Thibault, Namyst, and Wacrenier, Augonnet et al.) runtime system provides programmers with a portable interface for dynamically mapping tasks onto heterogeneous processors (CPUs and GPUs). It integrates development tuning and sampling with several pre-defined task scheduling strategies (National institute for research in computer science and control, 2008) as plugins. These include the eager scheduler that uses the minimum-completion-time-first policy (Topcuouglu et al., 2002), the dm scheduler that performs an HEFT-based scheduling policy, and several variations of the dm scheduler. Among all pre-defined schedulers, the best one is the dmdar (deque model data aware ready) scheduler. The dmdar scheduler is similar to the dm scheduler, but takes data transfer time into account and sorts tasks on a per-worker queue basis. Sc_hypervisor (Hugo et al., 2013) is an extension based on StarPU, which supports co-execution of multiple applications each using the StarPU runtime system. It focuses on partitioning approaches, which split computing resources into isolated sets, and then apply existing StarPU schedulers on each set. However, the StarPU runtime system does not focus on designing efficient mapping algorithms to minimize the completion time, but rather contributes in providing a portable interface for programmers to easily utilize GPUs. The StarPU pre-defined schedulers are mainly designed to handle the single application scenario and use simplified criterion to make mapping decisions.

2.3 GPU Resource Management

A number of runtime systems have been developed to perform GPU task scheduling. Qilin (Luk et al., 2009) provides an adaptive mapping to automatically partition tasks on a CPU and a GPU. StarPU (Augonnet, Thibault, Namyst, and Wacrenier, Augonnet et al.) runtime system provides programmers with a portable interface for dynamically mapping tasks onto heterogeneous processors (CPUs and GPUs). The aforementioned runtime systems are implemented at user-level and focus on heterogeneous systems without considering interference among multiple applications on the same GPU.

PTask (Rossbach et al., 2011) focuses on eliminating performance interference of GPU sharing. TimeGraph Kato et al. (2011) and Gdev (Kato et al., 2012) provide prioritization and isolation capabilities in GPU resource management. GDM (Wang et al., 2014) enhances GPU memory management by introducing a staging area in host memory for each process. These works are implemented at OS-level, and also propose scheduling algorithms for different applications sharing GPU resources. Specifically, Gdev and Timegraph make enhancements on kernel scheduling between contending applications. However, they can not handle the priority inversions caused by long-running or non-terminating kernels. Furthermore, GPES utilizes different interrupt schema compared to these work. More detailed differences between GPES and Gdev and TimeGraph are described in Sec. 3.2.3.

RGEM Kato et al. (2011) and PKM (Basaran and Kang, 2012) are two GPGPU engines which provide responsive and preemptive support for GPGPU tasks in a multi-tasking environment. However, they are implemented at user-level, thus lacking the view of the whole operating system. Moreover, in order to utilize their engine, GPGPU applications are required to be rewritten with the interfaces they provide. This may put much burden on end-programmers. Also, they have to know all applications before hand and then compile them into one single process, which does not reflect a real multi-tasking environment with dynamically coming applications. In contrast, GPES

does not need the source code of GPGPU applications. Also, it can transparently provide preemption and prioritization support for dynamically coming applications. To the best of our knowledge, GPES is the first piece of work which supports preemptive computation and memory-copying in a practical multi-tasking environment.

2.4 Real-time DNN-Based Object Recognition

DNNs have been extensively adopted in object detection for their impressive improvements in detection accuracy (Krizhevsky et al., 2012; Jia et al., 2014), which is the core function of many image/video processing applications. With GPU-accelerated platforms, DNN-based object recognition is now capable of processing vision workloads in real-time, either through algorithmic optimization (Redmon et al., 2016; Girshick et al., 2014; Girshick, 2015; Ren et al., 2015), or trading throughput with accuracy (Chen et al., 2015a; Han et al., 2016).

2.5 Optimizing Energy While Meeting Timeliness

The problem of optimizing energy while meeting hard or soft real-time constraints has received much attention in the literature (Farrell and Hoffmann, 2016; Hoffmann, 2015; Baek and Chilimbi, 2010; Sorber et al., 2007; Bini et al., 2009; Dudani et al., 2002; Heo et al., 2011; Huang et al., 2009; Imes et al., 2015; Hoffmann, 2014; Mishra et al., 2015). Despite various manipulation mechanisms in detail, these works mostly study simplified workload models (e.g., the well-studied sporadic independent task model (Mok, 1983) or approximate applications where accuracy can be traded for performance and/or energy).

A recent set of works (Farrell and Hoffmann, 2016; Hoffmann, 2015) have been proposed to explore the specific domain of approximate applications, where accuracy can be exploited for trading performance and energy. For instance, MEANTIME (Farrell and Hoffmann, 2016) seeks to minimize energy consumption while achieving timing predictability, with the core idea of trading

off accuracy for meeting deadlines. Other works such as Green (Baek and Chilimbi, 2010) and Eon (Sorber et al., 2007) seek to tailor behavior online to balance between accuracy and energy goals. CoAdapt (Hoffmann, 2014) allows users to prioritize two out of three goals in terms of performance, power, and accuracy, which then provides soft guarantees in those two prioritized dimensions while optimizing the third.

State-of-the-art research on optimizing energy efficiency for DNNs can be categorized into three categories: hardware approaches, DNN model optimization, and runtime approaches. Hardware approaches aim to optimize basic computations used in DNNs (e.g., convolution, matrix multiplication) (Chen et al., 2016; Han et al., 2016; Umuroglu et al., 2017; LiKamWa et al., 2016; Reagen et al., 2016; Chi et al., 2016; Shafiee et al., 2016; Albericio et al., 2016) through developing efficient hardware acceleration solutions. Model optimization approaches seek to compress or prune DNN models prior to execution (Han et al., 2015; Yang et al., 2017; Jaderberg et al., 2014; Kim et al., 2015; Romero et al., 2014; Xue et al., 2014; Chen et al., 2015; Han et al., 2015). Runtime approaches include offloading partial or entire DNN workloads to remote servers (Kang et al., 2017; Huynh et al., 2017; Bhattacharya and Lane, 2016; Chen et al., 2015b; Xu et al., 2017), and performing runtime approximation which trades performance with accuracy (Lane et al., 2016; Han et al., 2016). To the best of our knowledge, none of these works simultaneously consider timing correctness and energy efficiency by using native DVFS.

Different from all these works on optimizing latency and energy efficiency, PredJoule represents a system solution that can achieve timing predictability while minimizing energy for running DNN workloads. A unique contribution of PredJoule is to explore dramatically different performance/energy characteristics of DNNs through developing a layer-based approach. This allows the system to smartly identify the best configuration for running each layer, such that timing can be tightly controlled on a per-layer basis while achieving the most energy saving by considering each layer's performance/energy characteristics.

2.6 Kernel Transformation

Lee et al. propose SKMD (Lee et al., 2013) which transparently translates a single OpenCL (Group, 2008) kernel into variations and executes them on multiple GPUs simultaneously. Elastic Kernel (Pai et al., 2013) rewrites the kernel source code and reshapes the Kernel Grid to use $N : 1$ logical-to-physical mapping scheme. We implement kernel source transformation to slice a kernel into multiple subkernels and share the same idea of flattening workgroups as SKMD, but the goal of our technique is fundamentally different from those techniques: SKMD transforms kernels to distribute the workloads of a single kernel on multiple devices; Elastic Kernel uses kernel transformation to enable concurrent execution of different kernels; whereas we slice kernels to make the long-running kernel interruptible for better preemption. Furthermore, the source-to-source transformation technique is just a small (optional) part of our system, because we implement a novel and better kernel code rewriting technique as an alternative.

2.7 Managing GPUs in The Cloud

Current approaches for GPU management in the cloud are classified into I/O pass-through (AMAZON, 2006), API-remoting (Duato et al., 2010; Giunta et al., 2010; Lagar-Cavilla et al., 2007; Shi et al., 2012), para-virtualization (Dowty and Sugerman, 2009; Gottschlag et al., 2013; Suzuki et al., 2014) and full-virtualization (Suzuki et al., 2014; Tian et al., 2014; Malka et al., 2015; Zhou et al., 2015), the latter two being two different implementations of the device emulation technique. However, these works do not exploit the idea of GPU computing result reuse.

2.8 Computing Result Reuse

The concept of CPU-based computation reuse has been proposed in the programming language and computer architecture communities. Compiler-assisted approaches (Sodani and Sohi, 1997;

Conners and Hwu, 1999; Connors et al., 2000; Ding and Li, 2004) seek to reuse intermediate results at CPU instruction level. Function-level memoization (Michie, 1968; Pugh and Teitelbaum, 1989) is used to avoid re-executing functions by caching the results of prior function calls. Moreover, frameworks are proposed to reuse redundant computations at a higher level for the emerging incremental data processing field. For example, Spark (Zaharia et al., 2010), Percolator (Peng and Dabek, 2010), and CBP (Logothetis et al., 2010) provide programmers with facilities to store and reuse states across successive runs; while DryadInc (Popa et al., 2009), Nectar (Gunda et al., 2010), Haloop (Bu et al., 2010), Incoop (Bhatotia et al., 2011), CIEL (Murray et al., 2011), and Shredder (Bhatotia et al., 2012) are systems that reuse prior computing results. On GPU-incurred reuse, Arnau et al. (Arnau et al., 2014) presented a hardware memoization approach to eliminate redundant fragment shader executions on a mobile GPU. Different from these works, PredJoule focuses on GPGPU and efficiently realizing the partial GPU computing result reuse idea at a GPU kernel launch granularity.

CHAPTER 3

PREEMPTIVE EXECUTION FOR GPGPU[1]

3.1 A Case Study

Due to the asynchronous and non-preemptive nature of GPU processing, in multi-tasking environments, tasks with high priorities may be blocked by low-priority tasks. Such priority inversions may occur due to either kernel execution blocking or data transfer blocking. In a real-time system, this may cause deadline misses.

We conduct a measurements-based case study to show the impact of the non-preemptive kernel execution and data transfer blocking on real applications in practice, using two best available GPGPU drivers in a multi-tasking environment: the NVIDIA proprietary driver (NVIDIA, 2011), and the Nouveau open source driver (FREEDESKTOP, 2012) plus Gdev (Kato et al., 2012) which is a GPGPU run-time and resource management engine that manages GPUs as first-class computing resources. We measure the performance of running a video processing application *heartwall* (Che et al., 2009) competing with *mmul* (matrix multiplication). *Heartwall* processes a medical video frame by frame. A single frame processing consists of a memory-copying operation and a kernel launch. We assign the highest CPU priority to the *heartwall* application by viewing it as a high priority task, and assign low CPU priority to the *mmul* application as low priority task. The average processing time (memory-copy and kernel launch) of single iteration in *heartwall* is 380ms. It is set to execute periodically at an interval of 500ms. *Mmul* has variable processing time depending on its data size. It is configured to execute repeatedly with three sizes: small (256KB), medium (4MB) and large (16MB). Each combination executes for 500 seconds in total to impose high workloads on the entire system. We report the relative jitter and tardiness in the same manner

[1]©2015 IEEE. Reprinted, with permission, from Husheng Zhou, Guangmo Tong, and Cong Liu. "GPES: A Preemptive Execution System for GPGPU Computing", In Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS15). DOI:10.1109/RTAS.2015.7108420

Table 3.1: Jitter and tardiness of video processing application when competing with matrix multiplication under the NVIDIA proprietary driver (NV) and the Nouveau open source driver plus the Gdev module (Gdev)

	small	medium	large
NV tardiness	1169.64 ms	4387.7 ms	6130.77 ms
NV jitter	1188.13 ms	4523.22 ms	6483.90 ms
Gdev tardiness	426.24 ms	2678.97 ms	6726.78 ms
Gdev jitter	1197.11 ms	5095.33 ms	12705.99 ms

as (Kenna et al., 2011). Jitter is the deviation from the true periodicity of a periodic frame playback, which quantifies the smoothness of a video. If a frame i starts displaying at time t_i and the actual period between two frames is p, then its relative jitter is $|t_i - (t_{i-1} + p)|$. Tardiness represents the delay of completion.

As listed in Table 3.1, the jitter and tardiness on both drivers are significant, particularly when competing against *mmul* application with large data sizes. It is clear from this case study that current existing GPGPU drivers lack mechanism to make high-priority tasks preemptive when competing with long-freezing low-priority tasks. This lack of support motivates us to develop GPES, as described next.

3.2 System Design and Implementation

In this section we present the design and implementation of GPES, which aims to make tasks on GPU more preemptive and interruptible in multi-tasking environments. We implement GPES based on Gdev (Kato et al., 2012) which is open-source and publicly available. The software stack of GPES consists of a kernel transformer, a user-space library and an OS module. As depicted in Fig. 3.1, the shadowed rectangles represent the components of GPES. The kernel transformer performs automatic source-to-source transformation to kernel source code. The GPES library is a wrapper of driver APIs and provides CUDA API interfaces, where kernel execution slicing and data slicing are implemented. These two components are implemented at user space. The GPES

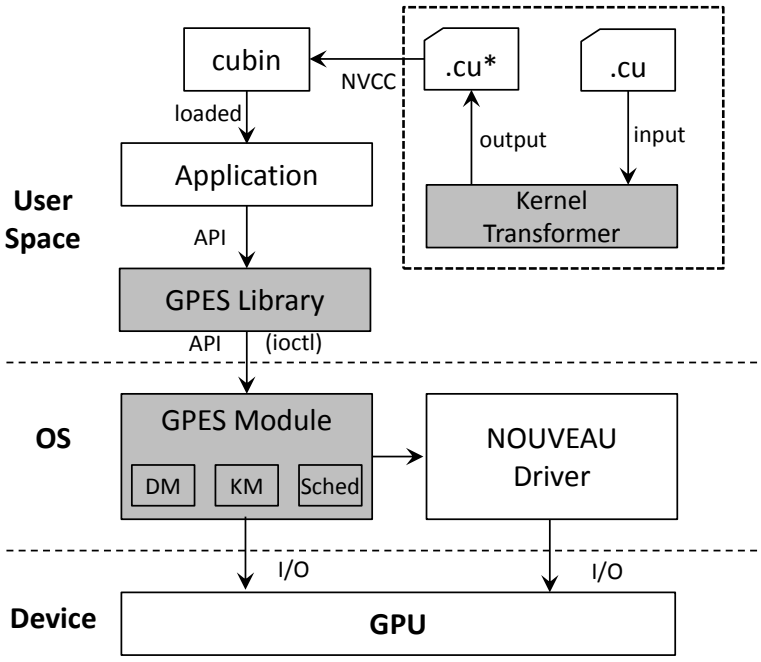


Figure 3.1: GPES framework

module performs the actual functionality of memory-copy, kernel launch, scheduling, and interrupt handling. GPES module is implemented at OS space.

GPES is implemented on top of the existing GPGPU programming framework of CUDA: source code of application is categorized into CPU code and GPU code; CPU code is compiled into executable file by gcc, whereas GPU code is compiled into object file (cubin) by nvcc (NVIDIA, 2011). The executable file executes on CPU and loads cubin file onto GPU. In the following sections, we highlight some of the implementation details that deserve articulation.

3.2.1 Kernel Execution Slicing

A long-running kernel can prevent other kernels from accessing GPU computing resources. To avoid this, one of our techniques is to slice the execution of a large kernel into smaller subkernels, so that high priority kernels can preempt control of the GPU after the completion of a subkernel. An ideal approach is to insert "preemption points" and control the mapping from blocks to SMs, where we force the GPU to execute part of the blocks each time. But unfortunately, currently the

CUDA programming model does not provide this level of controllability on SMs. The scheduler which dispatches logical blocks to hardware SMs is entirely implemented in the GPU hardware. NVIDIA has not disclosed the details of implementation to the public. A kernel is submitted in the form of a grid. Once a grid is offloaded to the GPU device, the execution is non-interruptible.

To achieve kernel execution slicing, we develop an alternative approach: workloads partitioning. In the following subsections, we introduce "source-to-source transformation" to better explain our idea and further introduce a novel technique "just in time kernel code rewriting" to make the kernel execution slicing totally transparent to applications.

Source-to-Source Transformation

To better support parallel computing, GPU hardware maintains continuous indexes for all blocks in one grid, i.e., *blockIdx*. For example, if a grid consists of 256 blocks in one dimension, the blockIdx ranges from 0 to 255. In order to make a long-running kernel interruptible, we convert a large kernel into multiple subkernels, each of which is launched with a *blockRange*. BlockRange is defined as the number of blocks to be executed in this subkernel, which is bounded by a start blockIdx and an end blockIdx. Slicing blockRange forces each subkernel to complete part of the computing workloads. Thus, the execution time of each subkernel is much shorter than the original kernel. At the end of each subkernel launch, we setup an "interrupt point" to allow higher priority kernels from other GPGPU applications to preempt the control of GPU.

Programmer can control the number and the shape of blocks in one grid. Such blocks are grouped up to three dimensions. For readability, here we consider for example a two-dimensional grid whose blocks are organized as a 16×16 rectangle. The blockIdx can be represented as a pair of (blockIdx.x, blockIdx.y), such as (0, 0), (1, 0), (2, 0). In order to assign a *blockRange* to each subkernel, our kernel transformation technique flattens N-dimensional blocks to one dimensional blocks, which makes the slicing easier. Fig. 3.2 shows that a two dimensional grid is flattened to

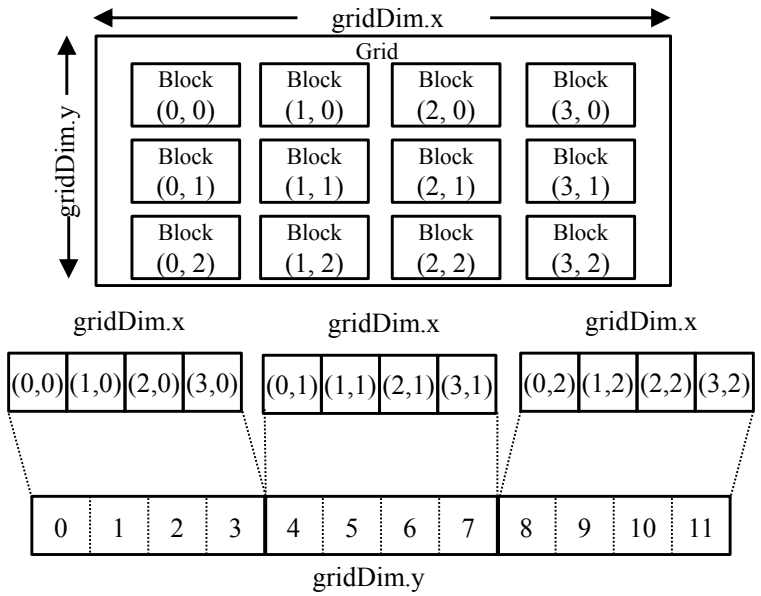


Figure 3.2: A two dimensional grid is flattened to one dimension

one dimension, where each block is associated with a flattened index. GPES executes an equal subset of blocks per subkernel launch, till all blocks are executed.

We implement kernel execution slicing technique through source-to-source transformation before the GPU code is compiled into cubin object file. This procedure is automatically performed by the kernel transformer. As shown in Fig. 3.3, the shadowed parts represent patched lines of code compared to the original kernel code in the *madd* (matrix addition) benchmark. Two parameters *block_from* and *block_to* are added to represent the range of flattened blocks to be executed. If a kernel launches more than one dimensional grid, lines of kernel-independent flattening code are inserted as shown on lines 5–7. After flattening the block indexes, each thread identifies its block index (flatId) and checks if it will continue to perform the computation. Thus, GPES actually performs computation of (*block_to*$-$ *block_from*$+1$) blocks for each subkernel launch and skips the rest of the computation. The computation of one kernel can thus be divided into several subkernels. Note that these patched lines of code are universal to all kernels and thus can be automatically applied by GPES.

```
1  __global__ void add(uint32_t *a, uint32_t *b,
2                      uint32_t *c, uint32_t n,
3                      int block_from, int block_to)
4  {
5      int flatId = gridDim.y * blockIdx.x + blockIdx.y;
6      if (flatId < block_from || flatId > block_to)
7          return;
8      int i = blockIdx.x * blockDim.x + threadIdx.x;
9      int j = blockIdx.y * blockDim.y + threadIdx.y;
10     if (i < n && j < n) {
11         int idx = i * n + j;
12         c[idx] = a[idx] + b[idx];
13     }
14 }
```

Figure 3.3: Kernel code transformation

To efficiently utilize the transformed subkernels, we re-implement several functions in the openCUDA (Kato, 2013) library. For example, we re-implement function *cuLaunchGrid* which is the CUDA API of launching a kernel, the pseudo-code is shown in Algorithm 1. The variable *SP* represents the number of subkernels that the original kernel will be sliced into, *range* denotes the number of blocks to be executed in each subkernel launch. Two additional arguments *block_from* and *block_to* are added to the subkernel's parameter buffer, which indicate the bounds of blocks. Function *cuLaunchGrid_v0* implements the actual subkernel launch. Till now, without touching the CPU source code, one kernel launch is converted to a number of subkernel launches specified by *SP*. Notice that, *SP* is a predefined value. Intuitively, a large SP value implies a small execution duration of each subkernel launch, which may also result in more overheads. The impact due to different *SP* values will be studied in Sec. 3.3.2.

Just In Time Kernel Code Rewriting

Source-to-source transformation must be completed at compile time. It has to access the GPU source file. Furthermore, once compilation is done, the granularity of partitioned block range cannot be changed at run-time. To transparently complete the kernel transformation without accessing the application's source code, we introduce a novel technique – "just in time kernel code

Algorithm 1 Customized cuLaunchGrid

```
1  cuLaunchGrid(f, grid_width, grid_height) {
2     range = grid_width * grid_height / SP;
3     rest = (grid_width * grid_height) % SP;
4     while(i < SP) {
5        block_from = i * range;
6        block_to = (i + 1) * range;
7        if ( i == SP - 1)
8           to += rest;
9        update_param_buf(f, block_from, block_to);
10       update_param_size(f);
11       cuLaunchGrid_v0(f, grid_width, grid_height);
12       i++;
13    }
14 }
```

rewriting". Through this approach, no source code of incoming applications is needed, as all transformations are performed transparently at run-time. Thus, it serves as a better alternative to replace the source-to-source transformation technique.

Through analyzing the changes between the generated GPU binary before and after the source-to-source transformation, we figure out that the additional instructions introduced in *mmul*'s kernel assembly code (i.e., SASS – shader assembly) are at the beginning, as shown in Fig. 3.4 lines 4–7. In order to do kernel binary code rewriting, two questions need to be solved: (i) how to insert the additional instructions into the original kernel code, and (ii) what impacts may such insertions introduce? Answering these questions is not easy, because NVIDIA does not disclose much of the details of its architectures and instructions. But fortunately, there are some third-party projects that have revealed useful information (Hou et al., 2011; Koscielnicki, 2012; Kato et al., 2012), such as the layout of GPU instructions, and the memory organization for a kernel launch. Currently two types of instructions are used in NVIDIA architecture: 4-byte and 8-byte. In our experiment environment, we always use 8-byte instructions. Each 8-byte GPU instruction is usually composed of na (specify the instruction name), mod (operation mode), pr (predicate bits), re0 (destination register), re1 (second register operand), imme (32-bit immediate value), and nb (specify instruction

name). To achieve the same goal of slicing kernel execution as source-to-source transformation, we must carefully follow this instruction format and insert our *range-selection* instructions.

To transparently rewrite the kernel code and reform the kernel, we add two parameters to the kernel (*block_from* and *block_to*) as discussed in the previous subsection. We re-implement *cuModuleLoad* and *cuModuleGetFunction* which are CUDA APIs loading cubin object file to get the kernel information (e.g., kernel code, kernel size, parameter size). After loading the GPU binary into memory, GPES reallocates a memory space for binary codes with additional 32 bytes for four *range-selection* instructions, and performs binary rewriting. When these four instructions are inserted, other instructions will be shifted. If there are unconditional branches, an offset-fixing action thus needs to be performed. For example at Fig. 3.4 line 16, the original branch destination is 0x108. Four additional instructions with 8 bytes each instruction causes 32 bytes (0x20) shift. For some complex kernels, we have to insert more than four *range-selection* instructions, since the registers used in inserted instructions may be further used for computation. In such cases, we need to introduce two more instructions to temporarily store and restore the register values. Additionally, GPUs maintain special registers for block indexes and parameters which are not explicitly revealed in SASS code. Thus, only modifying SASS code is not enough. As kernel's parameters are stored in constant memory, we have to modify the size of kernel's constant memory. The size of constant memory should be enlarged by 8 bytes since we add two additional int type parameters for block range-selection.

3.2.2 Data Transfer Slicing

Though data transfer and computation use different engines, memory copying of one application cannot perform simultaneously with kernel launching of another process, since they belong to different contexts and GPU can hold only one context at a time. Thus, in a multi-tasking environment, a large memory copy operation of a low-priority task can also stall high-priority tasks. To prevent

```
 1  MOV R1, c [0x1] [0x100];
 2  S2R R0, SR_CTAid_X;
 3  S2R R2, SR_CTAid_Y;
 4  IMAD.U32.U32 R3, R0, c [0x0] [0x18], R2;
 5  ISETP.GT.AND P0, pt, R3, c [0x0] [0x40], pt;
 6  ISETP.LT.OR P0, pt, R3, c [0x0] [0x3c], P0;
 7  @P0 EXIT;
 8  S2R R4, SR_Tid_Y;
 9  S2R R3, SR_Tid_X;
10  IMAD.U32.U32 R2, R2, c [0x0] [0xc], R4;
11  IMAD.U32.U32 R10, R0, c [0x0] [0x8], R3;
12  ISETP.LT.AND P0, pt, R2, c [0x0] [0x38], pt;
13  ISETP.LT.AND P0, pt, R10, c [0x0] [0x38], P0;
14  @!P0 EXIT;
15  ISETP.EQ.U32.AND P0, pt, RZ, c [0x0] [0x38], pt;
16  @P0 BRA 0x128;
    ... ...
```

Figure 3.4: Changes in GPU SASS due to kernel transformation.

this, GPES seeks to split a non-preemptive memory transfer into multiple smaller chunks to make it preemptive. At the boundary of the each chunk, a preemptive point is inserted.

Two CUDA APIs *cuMemcpyHtoD* and *cuMemcpyDtoH* are used to perform memory-copying between host memory and GPU device memory. To realize our idea, we re-implement these functions. Every single memory-copy operation is divided into multiple ones. Then each time GPES transfers only one chunk of data. At driver level, GPES is aware of all memory-copy requests from all user space applications. GPES maintains a queue of such memory-copy requests. Request with the highest priority will be put at the head of queue. Once the current memory-copy is done, the memory-copy request at the head of queue will be performed. GPES sets up a fence at the end of each transfer, the fence will raise an interrupt to notify the completion of the current transfer, and wake up the scheduled thread.

Intuitively, the chunk size impacts the granularity of preemption. The overhead introduced by fine-grained data transfer has been well studied by (Fujii et al., 2013; Kato et al., 2011). Our preliminary experiments showed similar results. Thus, in our implementation, if data chunk size in host-to-device memory-copying is no large than 4MB, we use direct I/O write; else we use DMA

engine to transfer data (according to (Fujii et al., 2013), (Kato et al., 2011), and our experiments). The threshold of device-to-host memory-copying is set to 4KB. The impact due to sliced chunks will be studied in Sec. 3.3.2.

3.2.3 Context Switch Scheduling

A GPU can hold only one context at any time. GPES uses interrupts to trigger context switches. GPES's context switch module is implemented at driver level, which uses two scheduler threads to perform computation scheduling and memory-copy scheduling separately. The scheduler threads are woken up by GPU interrupts generated upon the completion of any computation or memory-copy operation.

Different from TimeGraph (Kato et al., 2011) which is a GPU command scheduler integrated in GPU device driver to protect critical GPU applications from interference, GPES is API driven, which means interrupts are setup only when the interrupt function is called, and the scheduler is invoked only when computation or data transmission requests are submitted; while TimeGraph is command driven, interrupts are inserted in between command groups, causing the scheduler to be invoked whenever GPU commands are flushed. The scheduling overhead of GPES is thus much less.

Gdev is also API driven, and uses interrupts to invoke scheduler. Although GPES is implemented on top of Gdev, it uses very different interrupt schema to be more compatible with the kernel execution slicing and data slicing techniques. GPES setups interrupts for both kernel execution and memory-copying, which can make low-priority memory-copy operations yield to requests from other applications with higher priority; whereas Gdev does not use interrupts for memory-copy operations since Gdev synchronizes memory-copy with computation, causing the memory-copy operation to be non-preemptable. Furthermore, Gdev performs scheduling at the granularity of context-level. It creates a scheduling entity for each context. If the arriving scheduling entity

has the same context as the current entity, Gdev will not stall it. Gdev allows multiple continuous kernels that belong to the same context to be launched simultaneously in order to utilize the feature of concurrent kernel execution. However, such schema prevents the newly arriving high-priority tasks to preempt at the boundary between two such continuous kernels. Different from Gdev, GPES creates scheduling entity for each kernel launch and memory-copy chunk. If there is a scheduling entity occupying GPU, newly arriving schedule entities will be stalled regardless of whether it has the same context. Another difference between Gdev and GPES is that the priorities of GPU contexts are propagated from the OS to Gdev; whereas GPES not only supports such mechanisms but can also adaptively assign priorities to specific scheduling entities.

When one kernel is sliced into multiple subkernels, "interrupt points" inserted between subkernels will raise multiple interrupts. All interrupts from the GPU that are caught in the IRQ handler are relayed to GPES. When GPES receives an interrupt, it references the fence identity to verify which kernel launch or memory-copy operation raised the interrupt. At each interrupt point, the scheduling entity at the head of queue is popped out and set active, so that the application with the highest priority may preempt. The goal of the GPES scheduler is to correctly schedule computation and data transmissions for each GPU context based on priority.

3.2.4 Challenges and Limitations

Our GPES prototype implementation has several limitations. First, it does not yet support texture and 3-D processing. Thus when choosing benchmarks, we only choose CUDA-based image processing samples instead of OpenGL graphics. Another limitation of slicing kernel into multiple subkernels at block level is that we can not handle global synchronizations (if any) due to the global barrier originally deployed in the kernel. Furthermore, the research area of binary rewriting itself is still an open area. Our GPU binary rewriting technique can only handle simple kernels with simple semantics and a few unconditional branches. More sophisticated binary analysis techniques such as alias analysis, will be introduced to make the binary rewriting function more reliable. We leave such further improvements as future work.

Table 3.2: Benchmarks used in evaluation

NAME	Description	Structure
mmul	Matrix multiplication	Single kernel
madd	Matrix addition	Single kernel
heartwall	Medical imaging	1 kernel per loop
backprop	Back propagation	2 dependent kernels
bfs	Breadth-first search	Single kernel
hotspot	Physics simulation	1 kernel per loop
lud	LU Decomposition	3 dependent kernels per loop
nn	K-nearest neighbors	Single kernel
srad2	Image processing	2 kernels per loop

3.3 Evaluation

In this section, we present the experimental results used to evaluate the effectiveness of GPES.

3.3.1 Experimental Setup

Our experiments are conducted with the Linux kernel 3.3.1 on NVIDIA GeForce GTX 480 graphics card and Intel i7 4770K processor. Benchmarks are chosen from Gdev test samples and Rodinia benchmark suits (Che et al., 2009). Table 3.2 lists benchmarks used in experiments. All benchmarks are written in CUDA driver API and compiled by NVCC 4.0 (NVIDIA, 2011).

Because GPES focuses on the scheduling, it does not implement data swapping which is adopted in Gdev (Kato et al., 2012) as a core component to support excessive memory resource demands. We thus choose benchmark combinations that would not overload the GPU device memory. Furthermore, Gdev virtualizes a physical GPU into multiple logical GPUs, providing isolation and fairness among virtualized GPU devices. With different goals, GPES seeks to improve prioritization and preemptivity, and does not implement such virtualization. For fairness, we set the number of Gdev's virtual device to one and execute all benchmarks on this virtual GPU.

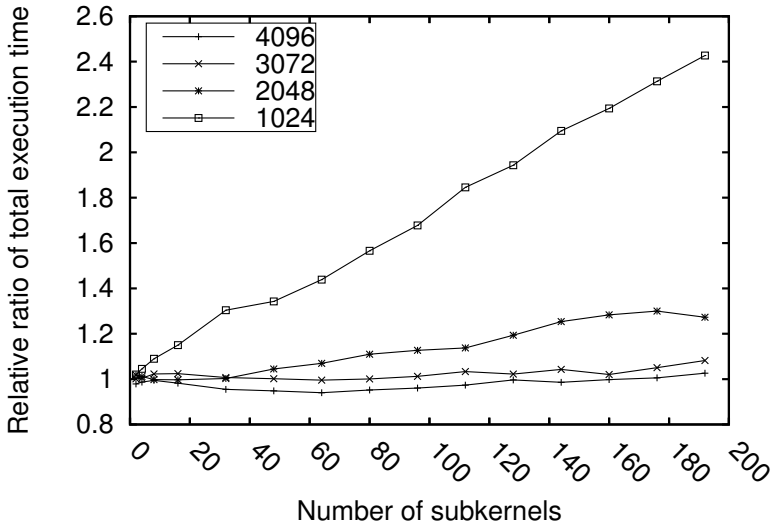


Figure 3.5: Relationship between execution time and number of subkernels

3.3.2 Overhead due to kernel slicing

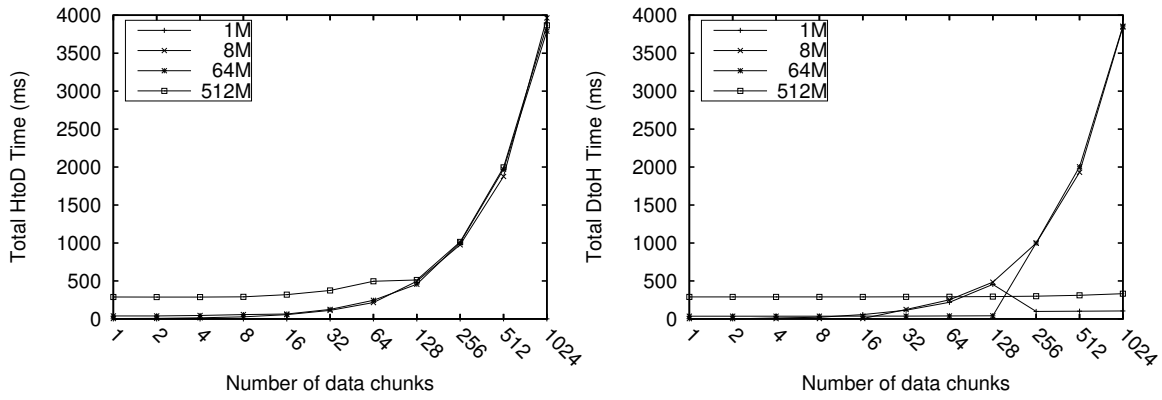
As mentioned in Sec. 3.2.1, one kernel launch can be sliced into multiple launches. Each launch offloads a subkernel with related parameters to GPU computing engine. Each launch of subkernel will introduce overheads caused by initialization and hardware dispatching. Also, each thread will execute at least four additional instructions to identify its *blockIdx*. All these extra works will introduce overheads.

In order to evaluate the performance of our implementation of kernel slicing, we slice the kernel of *mmul* benchmark into multiple subkernels, and record its execution time. As shown in Fig. 3.5, the x-axis is the number of subkernels that we slice the original kernel into; the y-axis is the relative ratio of the total execution time with slicing divided by the execution time of the original kernel without slicing (abbreviated as *relative ratio*); four curves in this figure represent *mmul* with 4 different input sizes (e.g., 1024×1024). We observe from Fig. 3.5 that kernel slicing introduces reasonable amount of overheads. For example, when the number of subkernels reaches 400, the *relative ratio* of *mmul* instance with 4096×4096 input size is only 1.04. We also observe that instances with larger inputs tolerate more fine-grained slicing. Essentially, there is a tradeoff between the granularity of kernel slicing and overhead, which often depends on different

applications and devices. To accurately identify an appropriate value of subkernel granularity, an offline profiling is needed. However, GPES makes kernel slicing decisions online to handle dynamically coming applications. In our experiments, we pick the number of blocks executed in each subkernel launch to be no less than 120, since with such subkernel granularity, the slowdown of all benchmarks introduced by kernel slicing can be limited to less than 5%. This is reasonable because our GTX 480 GPU can hold at most 120 blocks simultaneously. If the number of blocks to be executed on GPU is less than 120, the computation resources may not be fully utilized. In summary, the kernel slicing technique adopted by GPES is efficient and applicable, with acceptable overhead.

3.3.3 Overhead due to data slicing

We also measured overheads caused by our implementation of data slicing using the *memcpy* benchmark. The *memcpy* benchmark performs memory-copying from host to device and then transfers the data back using DMA without doing any kernel computation. Fig. 3.6 (a) illustrates the impact of data slicing on host-to-device (HtoD) memory-copying time. The x-axis is the number of chunks, y-axis is the total HtoD time from the first chunk to last chunk. The curves represent *memcpy* instances with different input data. We observe that the total HtoD time increases with the increased chunk number. However, such increases are reasonably small when the number of chunks is no greater than 128. When the number of chunks is greater than 128, the HtoD time substantially increases. This is because the overhead introduced by using the DMA engine is non-trivial. Fig. 3.6 (b) reveals the same trend as in set (a) on the device to host memory-copying time. We observe that the total DtoH time of instances with smaller inputs (1M, 8M, 64M) increases with the increased number of chunks. But when the 1M instance is sliced into 256 chunks (4KB per chunk), it stops increasing. This is because it hits the threshold of 4KB where direct I/O performs better than DMA engine when transferring data with size less than 4KB from device to host. However, such fine-grained slicing is not encouraged since it introduces non-trivial overhead compared



(a) (b)

Figure 3.6: Relationship between memory-copy time and number of chunks. (a) Host to device (b) Device to host

to memory-copy without slicing. Meanwhile, the instances with 512MB input performs almost consistently regardless of the growth of the number of chunks, since large chunks can benefit from DMA transaction. With 1024 chunks and 512MB input data, the total DtoH time with slicing (310ms) is only 6.8% more than the time without slicing (290ms). In the experiments, we pick the HtoD chunk size to be no less than 4MB and the DtoH chunk size to be no less than 512KB, since such granularity provides good tradeoff between fine grained preemption and overhead.

3.3.4 Overhead of context switching

There are two stages that would introduce context switching overheads in computation scheduling. One happens at each new kernel launching time, when GPES locks the computation scheduling thread, then performs context switch scheduling, and finally unlocks the computation thread. During this stage, GPES checks the current status of GPU. If the previously offloaded kernel is not returned, GPES will stall the launch request; otherwise it sets the incoming context active. The other stage happens when an interrupt indicating kernel completion is caught. Between the locking and unlocking operations to the computation scheduling thread, GPES draws out the context with the highest priority in the request queue and sets it to active. Similarly, there are also two stages which introduce context switching overheads in scheduling memory-copying operations.

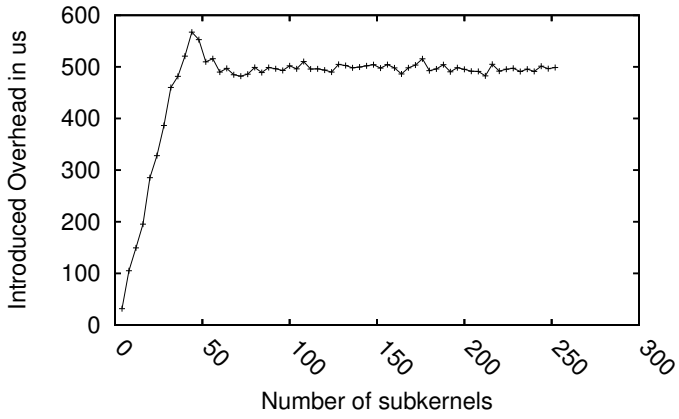


Figure 3.7: Additional context switch overhead via kernel execution slicing

We evaluate the context switching overhead in GPES by recording the time slots when scheduling threads are suspended. We use the *mmul* benchmark with 16MB input to evaluate the context switching overhead caused by kernel slicing, as illustrated in Fig. 3.7. The x-axis is the number of subkernels, the y-axis is the overhead introduced by context switch scheduling in μs. We observe that the overhead increases with the increased number of subkernels till the number reaches 50 subkernels, after which the overhead stops increasing. The reason is that with a large number of subkernels, each subkernel executes very fast such that no incoming requests will be stalled. Thus, no additional overhead will be introduced. The result of context switching overhead caused by data slicing is similar to Fig. 3.7. Thus, we conclude that our context switching overhead is trivial (less than 600 μs) compared to the execution time of kernel launch and memory-copying.

3.3.5 Multi-Tasking Performance

In this subsection, we discuss the performance of high-priority applications when competing with low-priority applications under GPES. Following are some terms that will be used in the description.

Response time: The time elapsed between the context creation and context destruction. We recorded this time by measuring the time slots between *cuCtxCreate* and *cuCtxDestroy* that are called by an application.

Compute-Occupying time: The *compute-occupying time* of a specific kernel is the number of time slots when it is offloaded to GPU till its interrupt indicating completion is captured by GPES. The *compute-occupying time* of a CUDA application is the sum of all its kernels.

Copy-Occupying time: The time period that an application occupies the GPU copy engine.

Occupying time: Sum of *compute-occupying time* and *copy-occupying time*.

Pending time: The difference between *response time* and *occupying time*.

Impact of kernel execution slicing. In the first set of two experiments shown in Fig. 3.8, we evaluate the performance of high-priority applications with different kernel structures when competing with low-priority computation-intensive applications under GPES and Gdev. Benchmark *mmul* is chosen as the low-priority application (LP) in all these three experiments, which is considered as computation-intensive application. GPES only performs kernel execution slicing since the memory-copying time is relatively small compared to kernel execution time. We execute two instances of *mmul* with the same input size and priority repeatedly to interfere with high-priority applications.

In the first experiment, we choose *madd, bfs, nn* with small input sizes as high-priority applications (HP). These benchmarks are all single-kernel applications. They execute periodically with an interval of 5,000ms. This configuration can avoid the interference among high-priority tasks whereas each high-priority task can compete with at least one low-priority task. We report the average pending time of the three high-priority applications in Fig. 3.8 (a). The x-axis is the input data size of LP; the y-axis is the relative ratio of HPs' average pending time divided by the standalone execution time of LP without kernel execution slicing, denoted as normalized average pending time; the four curves represent Gdev, GPES with each kernel sliced into two subkernels (GPES+2SP, for short), GPES with eight subkernels (GPES+8SP), GPES with 32 subkernels (GPES+32SP). We observe that with our kernel execution slicing technique, the average pending time is reduced dramatically compared to Gdev under all scenarios. For example, when the input data of LP reaches 100MB, the normalized average pending time of Gdev is 0.98, while the nor-

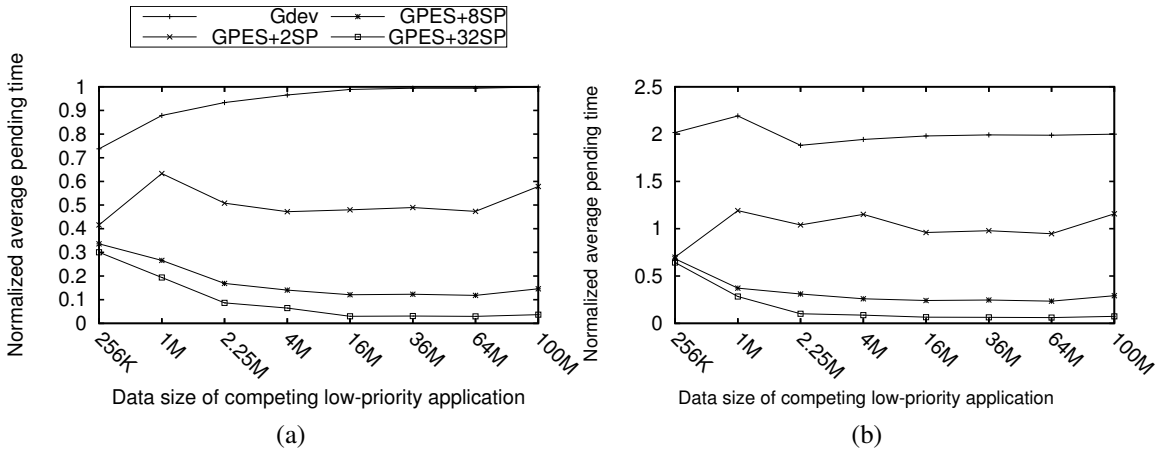


Figure 3.8: Impact of kernel execution slicing. (a) Single kernel (b) Dependent kernels

malized average pending time of GPES with 32 subkernels is less than 0.05, which is more than 90% reduction.

In the second experiment, we choose *backpro, heartwall, lud* as HPs. All these benchmarks have dependent kernels, which means that the second kernel launch must be performed after the first launch. There is a memory-copying operation between two kernel launches. As seen in Fig. 3.8 (b), we observe that the average pending time is almost doubled compared to the previous case (Fig. 3.8 (a)). For example, when being interfered by LPs with 100MB input, the normalized average pending time of Gdev and GPES with two subkernels are 2.0 and 1.2 respectively. The reason is that the two dependent kernels of HP are interleaved by a subkernel of LP, causing extra pending time. Nonetheless, GPES significantly outperforms Gdev in reducing the average pending time of HPs.

Impact of data slicing. In the second set of three experiments, we evaluate the impact of data slicing on applications when competing with low-priority data-intensive applications under GPES and Gdev. Benchmark *memcpy* is used as the competing LP, which does not contain any kernel launch. Three sets of benchmarks are chosen as HPs: computation-intensive set, data-intensive set, and mixed set. LP is configured to execute repeatedly, while each instance of HP executes for every 5,000ms to avoid the interference among high-priority tasks and make sure that each

high-priority task can compete with the low-priority task. The results of these three experiments are depicted in Fig. 3.9, where x-axis is the input data size of LP; the y-axis is the relative ratio of HPs' average pending time divided by the standalone execution time of LP without data execution slicing, denoted as normalized average pending time; the four curves represent Gdev, GPES with each memory-copying data sliced into four chunks (GPES+4CK), GPES with 32 chunks (GPES+32CK), GPES with 256 chunks (GPES+256CK).

In the first experiment, we choose *mmul, lud, heartwall* as HPs. These benchmarks are all computation-intensive applications. Heartwall is configured with only one iteration; mmul is configured with 2048×2048 matrix; lud is configured with 1024×1024 matrix. We observe from Fig. 3.9 (a) that the data slicing technique is efficient to reduce the average pending time in all scenarios compared to Gdev. For example, when the data size of LP reaches 512MB, the average pending time with Gdev is 0.63, whereas GPES with 256 chunks is 0.25.

In the second experiment, we use *nn, backpro, bfs* configured with large data size input as HPs. These benchmarks are data-intensive applications. Specifically, we modify nn to read all data from one data file instead of originally from thousands of data files, in order to reduce the file I/O latency. We observe from Fig. 3.9 (b) that the average pending time is greater compared to (a), which is caused by the fact that large data transfers often cause longer-time GPU initialization (e.g., memory allocating) which is non-preemptive. Nonetheless, the data slicing technique of GPES is still very efficient in reducing the average pending time in all scenarios compared to Gdev.

In the third experiment, we mix all the six benchmarks used in the previous two experiments together with the same execution configuration. We observe from Fig. 3.9 (c) that GPES is still superior to Gdev.

Video case study. Recall the case study used in Sec. 3.1, the jitter and tardiness increase rapidly under NVIDIA proprietary driver and Gdev. For the same video processing application, we conduct an evaluation using GPES to slice host-to-device memory-copying into 4MB chunks and device-to-host memory-copying into 512KB chunks, and slice kernel into 120 blocks per sub-kernel launch. The result is shown in Fig. 3.10. We observe that the jitter and tardiness can be

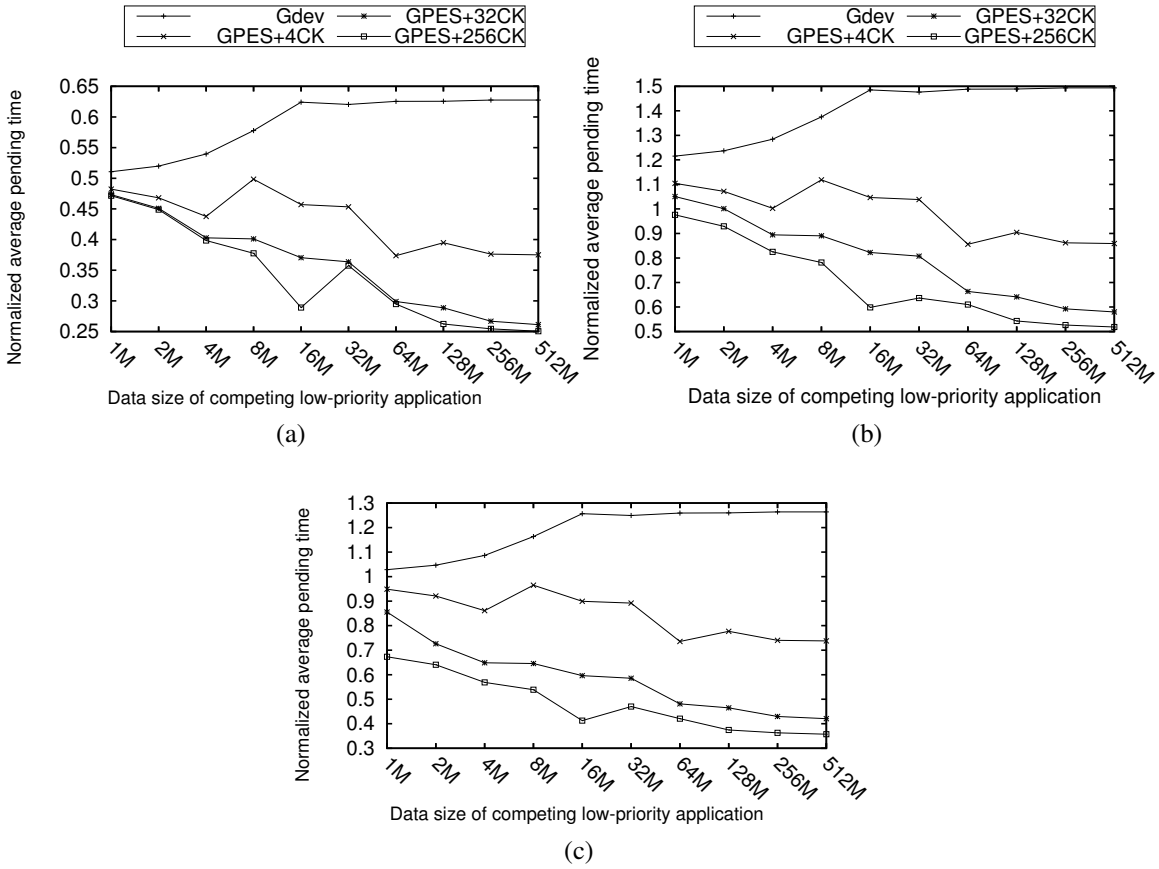


Figure 3.9: Impact of data slicing (a) Computation-intensive (b) Data-intensive (c) Mixed

significantly reduced compared to Gdev. For example, when the data size of competing application reaches 36MB, the tardiness and jitter under Gdev are 11,253ms and 21,398ms respectively; whereas GPES is able to reduce these values to 1,388ms and 2,580ms. The jitter and tardiness under GPES are much lower than Gdev in all cases, and the reduction can reach up to 80%. GPES can thus prevent video applications from being interfered by low-priority competing applications containing long-freezing memory-copying or kernel execution.

3.3.6 Non-real-time setting

For applications that do not have predefined priorities, GPES can still reduce the overall pending time. We conduct an evaluation using three benchmarks: mmul (1024×1024), srad (35 iterations),

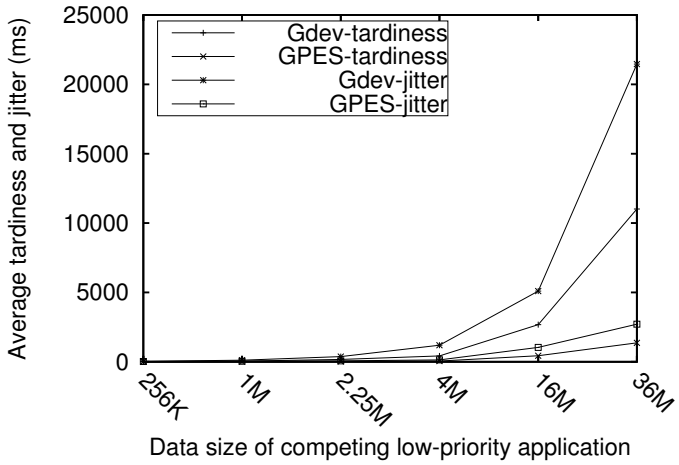


Figure 3.10: Jitter and tardiness of image processing case under Gdev and GPES.

nn (file size 1024). These three benchmarks execute in different orders which reflects different priorities under Gdev and GPES. The overall pending times are depicted in Fig. 3.11. The x-axis indicates the combination of the three benchmarks. For example, mmul.srad.nn indicates mmul is firstly loaded, srad is the second and so on. We observe that the overall pending time in most scenarios under GPES is much less than Gdev. For example, in the combination of mmul.srad.nn, the overall pending time under GPES is 53% less than Gdev. However, as shown in the last bar of Fig. 3.11, the overall pending time under GPES is larger than Gdev. This is because it represents the best ordering under both GPES and Gdev. But in this case, GPES introduces extra overhead due to slicing.

3.3.7 Defending against DOS Attacks

In many systems, a malicious GPGPU application can attack the system by submitting large numbers of kernels or an extremely large kernel which consists of many threads, causing denial-of-service (DOS) to normal GPGPU applications. GPES can mitigate such attack. By enforcing slicing large kernel with a lot of threads into smaller ones and each time executing a range of threads, GPU control can be regained by normal applications when an interrupt is issued.

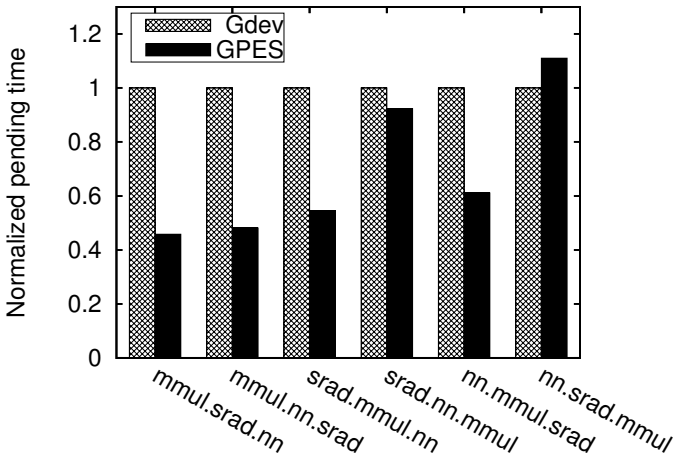
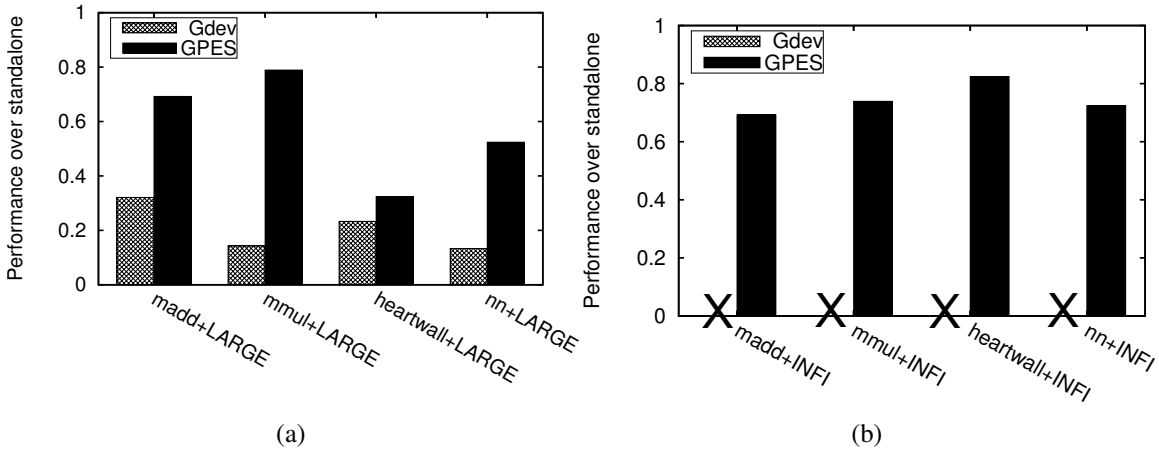


Figure 3.11: Normalized pending time under Gdev and GPES.

In order to demonstrate how GPES mitigates such DOS attack, we have hand-coded two malicious GPGPU applications: LARGE and INFI. LARGE is a malicious application with a very simple kernel but consisting of a large number of threads. INFI is a malicious application issuing kernels repeatedly. We co-run each of our benchmarks (heartwall, madd, mmul, hotspot, nn) with LARGE and INFI, and compare the makespan under GPES and Gdev. We execute the malicious application first, and then start running our benchmarks. Makespan is the elaspe from the benchmark's start to its completion. The results are shown in Fig. 3.12, where the y-axis denotes the ratio of the makespan of running the benchmark alone without any malicious applications divided by the makespan with malicious applications. With GPES, the LARGE is sliced into 1024 subkernels, and all benchmarks successfully complete execution in acceptable time, averagely slowed down by 49% compared to standalone executions. With Gdev, all benchmarks are slowed down by 90% when co-running with LARGE or 100% (non-terminated) when co-running with INFI.

3.4 Summary

In this chapter, we present GPES, a GPGPU preemptive execution system to make long-running low-priority GPGPU applications interruptible and preemptable in a multi-tasking environment.



(a) (b)

Figure 3.12: Defending against malicious applications (a) LARGE (b) INFI. (An 'X' mark means that the normal application does not terminate and the performance cannot be measured.)

We implement a prototype system based on open-source GPGPU drivers. Our system introduces several techniques that slice a long-running kernel into several smaller subkernels and slice data transmissions into chunks with acceptable overheads. In order to achieve better preemptivity, GPES also implements new interrupt handling and context switching schemes. Experimental results demonstrate that GPES can achieve much better performance compared to the state-of-art open-source driver, and performs consistently well across different applications. The incurred overheads due to the proposed techniques are reasonably small, which makes GPES a practical and efficient solution for real-time GPGPU computing.

CHAPTER 4

STREAM SCHEDULING FOR GPU-ACCELERATED REAL-TIME DNN WORKLOADS[1]

4.1 Motivation

In this section, we illustrate a set of measurements-based case studies to motivate our design. We use YOLO (Redmon et al., 2016) as the target program, representing the state-of-the-art, real-time DNN-based object detection frameworks. YOLO is capable of taking continuous frames from a video file or a camera as input, and is able to output frames with labeled objects. We run multiple YOLO instances simultaneously on an NVIDIA Quadro 6000 GPU, each of which uses a road drive video from the KITTI vision benchmark suite (Fritsch et al., 2013) as input. We use average processing FPS (frames per second) of all processed frames to measure the throughput of the system. If the object recognition programs can process every frame of the video at a higher or equal FPS than the original processing time of the video (i.e., 40 ms per frame), we consider that the video can be processed in real-time. Besides average FPS, we also use deadline miss rate, denoted $pMiss$, to indicate the percentage of frames that miss the deadline. We note that, different from online cameras, pre-recorded videos can be processed at a higher FPS than the origin, since such videos are actually processed as a series of images – once the previous image has finished being processed, the next image is processed immediately afterwards without waiting for the release time. We also consider online cameras in the evaluation (Sec. 4.3).

4.1.1 GPU Usage Pattern For DNNs

In the first case study, we directly use YOLO to perform object detection on an input video. The neural network is configured to use the default setup (yolo.cfg), which is composed of 16 layers

[1]©2018 IEEE. Reprinted, with permission, from Husheng Zhou, Soroush Bateni, and Cong Liu. "S3DNN: Supervised Streaming and Scheduling for GPU-accelerated Real-Time DNN Workloads", In Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS18). DOI:10.1109/RTAS.2018.00028

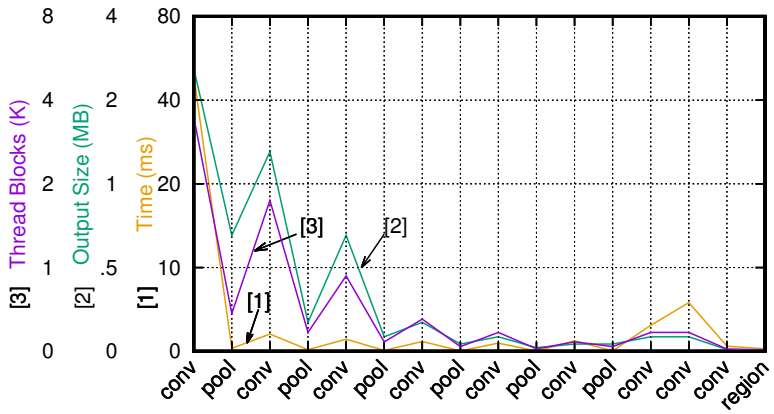


Figure 4.1: Resource usage pattern of DNN workloads.

of three types depicted on the x-axis of Fig. 4.1 ("conv" for the convolution layer, "pool" for the max-pool layer, and "region" for the region layer). For each layer, the y-axis reports the size of the output which is temporarily stored in GPU global memory (in MB), number of thread blocks (in Kilo), and execution time (in ms) in processing each frame. Two key observations can be obtained from this figure: (1) The GPU resource utilization, including both the global memory usage and the thread block numbers, shows a "staged" pattern as the layers go deeper (i.e., consistent gradual decrease on resource utilization along with the increased depth of layers). (2) The final layers exhibit a small input data and light computation, thus may under-utilize the GPU resources. The intuition behind these observations is that DNN generates more fine-grained feature maps at earlier layers and prunes these details gradually along the depth increase to get a higher level of abstraction. In classic DNN models (Krizhevsky et al., 2012; Szegedy et al., 2015; Simonyan and Zisserman, 2014), they usually use fully connected layers at the last few layers to match the pattern generated from input images with all the possible classes stored in the model file. The input size of the fully connected layer is determined by the number of classes. In the case of real-time object detection in autonomous driving which is the focus of this dissertation, the class number is usually small since we are only interested in a small subset of objects (e.g., pedestrian, stop sign, vehicle, traffic signal). For example, the class number in our experiments provided by vanilla YOLO is 20 and 80, trained from the coco dataset (Lin, Maire, Belongie,

Bourdev, Girshick, Hays, Perona, Ramanan, and Piotr Dollár and C. Lawrence Zitnick, Lin et al.) and voc (Everingham et al., 2010) respectively. We note that we have observed similar trends on other popular DNN-based object detection tools (Girshick et al., 2014; Girshick, 2015; Ren et al., 2015) using their default configurations.

Insight 1: The GPU usage pattern in DNN shows a staged GPU resource utilization pattern, where the earlier layers involve more intensive computations and larger input sizes, and the later layers incur lighter computations and smaller input sizes, which may under-utilize GPU hardware.

4.1.2 Data Fusion

Fetching multiple images in a "batch" to process them in one pass is a common optimization method used in DNNs, which can significantly improve the throughput with a proper batch size (Krizhevsky et al., 2012; Simonyan and Zisserman, 2014; Szegedy et al., 2015). The improvement is due to the fact that using batch can reduce the time wasted in I/O operations and data communication round-trips, and thus can improve performance compared to processing images sequentially. Moreover, in some cases, it is able to improve GPU utilization.

Application Level Data Fusion

Many prominent DNN implementations such as Caffe (Jia et al., 2014) include a data fusion functionality. For a multi-tasking environment where multiple DNN instances are running simultaneously, it is possible to batch multiple images into one instance instead in order to improve overall throughput. This approach is particularly useful under the autonomous driving scenario, where a vehicle is often equipped with multiple cameras and sensors to cover front, sides, and rear view. In this case study, we illustrate the effect of data fusion on throughput and latency.

We modified YOLO to support batch-based processing of up to four videos in order to process them all at once. The baseline uses up to four vanilla YOLO programs, each of which processes one

input video. These videos are of the same target FPS (25 FPS). We use average FPS to measure the throughput in two different configurations, baseline without fusion and our implementation with data fusion which fuses all input frames together. To measure the real-time performance, we use *pMiss* to record the percentage of frames that miss their deadline. The results are shown in Table 4.1, where the first row indicates the number of incoming video streams, the second and third rows show the average FPS, and the fourth and fifth rows record the *pMiss*.

We observe that, with more incoming videos, data fusion becomes more efficient in improving throughput, as indicated by the average FPS. Another observation is that data fusion causes a 100% pMiss ratio when fusing four video streams together, yet achieving a high FPS (21.2×4). This is because, with data fusion, all the fused images will start and complete at the same time with a longer processing time compared to the individual execution scenario. In this case study, when fusing four video streams together, the resulting processing time exceeds the deadline of each frame, thus incurring a 100% pMiss ratio. In practice, multiple cameras on an autonomous driving vehicle may be of different FPS. For example, cameras on the dashboard may have higher FPS than cameras at the rear end. In such scenarios, fusing data from cameras with different frequency may lead to more frequent deadline misses for cameras with higher FPS.

System Level Solution

While existing application level data fusion has proven effective in increasing throughput, it falls short on multiple accounts. First and foremost, for a multitasking environment consisting of multiple DNNs (such as in autonomous driving), application level data fusion will not be effective since it would not be aware of other DNN processes in the system. Moreover, in such systems the number of executing DNN instances varies from time to time. Since application level batching needs to be pre-configured, it would not be able to adjust. For example, in Table 4.1, the data fusion FPS can vary significantly depending on the number of videos and can cause many variations that a real-time system needs to deal with. Finally, an application level solution would not be able

Table 4.1: APT and pMiss of data fusion and base line

# of videos	1	2	3	4
Baseline FPS per video	52.6	29.2	18.5	15.6
Data fusion FPS per video	52.6	33.3	26.7	21.2
Baseline *pMiss*	0%	5%	83%	92%
Data fusion *pMiss*	0%	0%	0%	100%

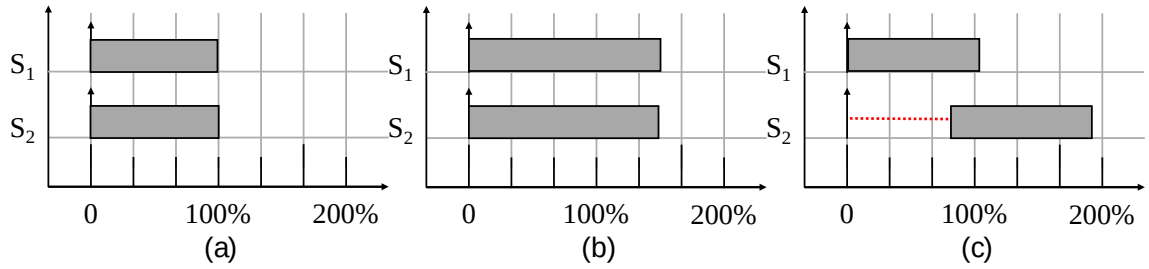


Figure 4.2: Execution time of two streamed concurrent Kernels with different numbers of thread blocks: (a) small number (b) medium number (c) large number.

to recognize data priority and take scheduling decisions into consideration, which might result in lower-priority tasks to be batched with higher-priority tasks, causing an unfair and potentially dangerous situation. This is evident in the 100% miss rate visible in the 4 video data fusion as shown in Table 4.1.

Insight 2: To overcome the under-utilization issue incurred by individual DNN workloads, fusing data from multiple videos may achieve significant throughput gain yet may harm the per-frame response time for each individual video.

4.1.3 Kernel Scheduling and Concurrency

Besides data fusion, scheduling tasks in an orderly fashion while utilizing the CUDA stream technique may also improve throughput by enabling concurrent execution of multiple kernels bound to different streams while ensuring deadlines, as illustrated by the following case study.

Enabling Concurrency using CUDA Streams

To understand how GPU performs concurrency for kernels in different streams, we conduct three case studies, each of which performs matrix multiplication-based computation that is frequently used in DNN. In all the case studies, we concurrently launch two kernels placed in two different CUDA streams. The only difference among three case studies is the total number of thread blocks contained in each kernel. The execution traces are shown in Fig. 4.2, where case studies (a), (b), and (c) have 7, 28, and 500 thread blocks in each kernel, respectively. The x-axis represents the normalized execution time defined by the actual execution time divided by the execution time of executing only one of the two kernels under each scenario.

We observe in Fig. 4.2(a) that the response time of completing two kernels is the same as only one kernel being executed, thus enabling a perfect concurrency for this scenario. This is because the total number of thread blocks contained in these two kernels equals the number of SMs in GPU hardware (i.e., 14 thread blocks or SMs). Thus, each thread block is assigned to a dedicated SM. In Fig. 4.2(b), the observation is that the response time becomes longer than the standalone execution time of one kernel. In this case, although the total number of thread blocks (56 in this case) is more than the number of SMs, concurrency can still be enabled because NVIDIA compute capability 2.x devices can support up to 8 blocks per SM. However, due to the large number of thread blocks, these thread blocks from two kernels will be executed in an interleaving manner, causing a performance better than serialized execution but worse than the perfect concurrent execution case (e.g., Fig. 4.2(a)). For the third case study where we aggressively increase the total number of thread blocks to 1000, as seen in Fig. 4.2(c), we observe that these two kernels can only partially overlap to a rather limited degree of concurrent execution. The reason is because with a large number of thread blocks per kernel, the first kernel will fully occupy all GPU resources when it starts execution. Near the end of the first kernel's execution, there are a remaining 52 thread blocks (i.e., 52 = 500 thread blocks mods 14x8 GPU capacity) to be executed on GPU, the second kernel can start execution concurrently using the available SMs on GPU.

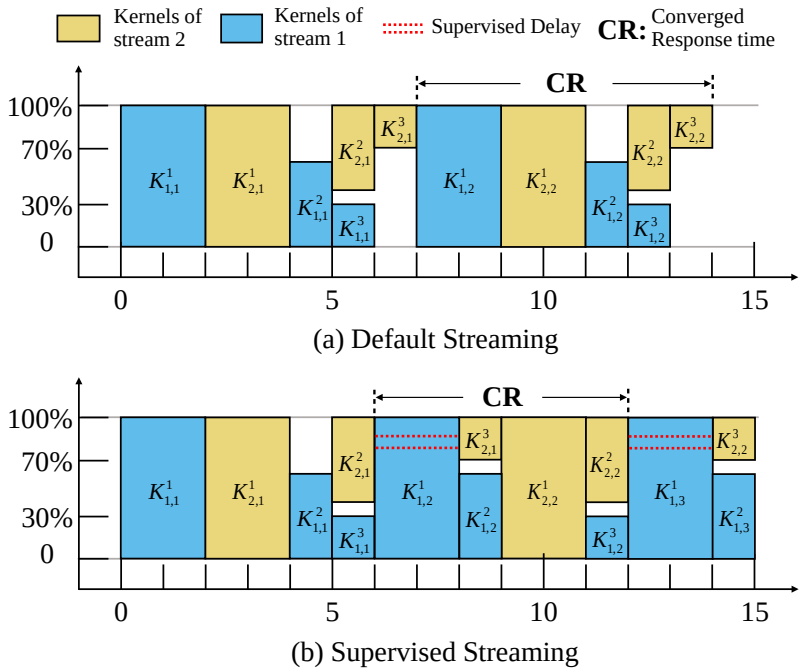


Figure 4.3: Concurrency under CUDA stream without supervised streaming (inset (a)) and with supervised streaming (inset (b)).

Insight 3: Concurrent kernel execution through putting each kernel in a different CUDA stream may improve overall system throughput performance, particularly when individual kernels cannot fully utilize GPU resources. However, adopting CUDA stream to improve concurrency may harm the timing predictability of individual kernels due to interference.

Supervised CUDA Streams with Scheduling

Based upon Insight 3, we perform another case study to understand how DNN workloads may further benefit from stream-enabled concurrency. We run two identical periodic DNN tasks, each consisting of three layers. To enable concurrency, we bind each task to a separate CUDA stream. We configure the first, second and third layer of each DNN task to occupy 100%, 60% and 30% of the GPU resource, i.e., 224, 67 and 34 respectively, thread blocks according to the used GPU hardware. Fig. 4.3(a) shows the execution schedule constructed from the measurements data when we run two streams concurrently. Let $K^k_{i,j}$ denote the k^{th} layer/kernel of the j^{th} job (i.e., the

j^{th} video frame) released by DNN task K_i. As seen in Fig. 4.3(a), streaming indeed improves performance by enabling concurrent execution. For example, $K^2_{2,1}$ is concurrently executed with $K^3_{1,1}$ since the total number of thread blocks of these two kernels is smaller than the available thread blocks supported by the GPU hardware.

While we are seeking to further improve concurrency, a very interesting observation appears. As seen in Fig. 4.3(a), $K^3_{2,1}$ is executed alone without other concurrent kernels. However, performance may be improved by concurrently executing $K^3_{2,1}$ with a later released kernel $K^2_{1,2}$. To achieve this, we have to let $K^1_{1,2}$ execute first and preempt $K^3_{2,1}$, thus releasing $K^2_{1,2}$ upon its completion. The resulting execution schedule is shown in Fig. 4.3(b), where clearly both response time performance and concurrency are improved.

Insight 4: To optimize concurrency for DNN workloads exhibiting staged computation demand, instead of performing default CUDA streaming, it may be much more beneficial to perform "supervised streaming", which judiciously schedules kernels for maximum concurrency benefits through considering varying resource requirements of different layers of DNN tasks.

4.2 Design and Implementation of S^3DNN

4.2.1 Design Overview

S^3DNN is designed to serve as a middle-ware between input videos and GPU hardware to optimize the execution of DNN-based object detection workloads on GPU. S^3DNN is implemented as a frontend-backend framework, where multiple S^3DNN frontends take videos as input and forward all data and DNN processing requests to the S^3DNN backend for actual computation, as shown in Fig. 4.4.

The S^3DNN backend consists of two major components, a *Governor* (Sec. 4.2.2) and an S^3 *scheduler* (Sec. 4.2.3), motivated by the four insights discussed in Sec. 4.1. Whenever multiple

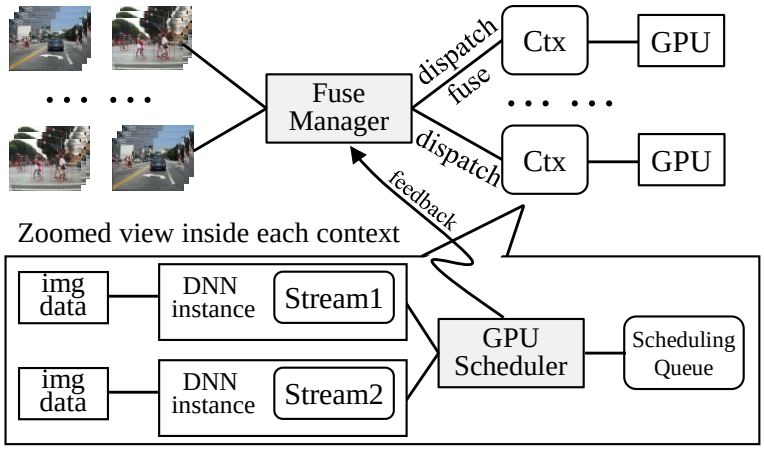


Figure 4.4: Design overview of S^3DNN.

video frame processing requests are forwarded to the backend, the governor will decide to selectively fuse them together so that they can be processed in fewer *DNN instances*. The governor also assigns the fused DNN instances to one or several virtual contexts, each of which is associated with a GPU device according to the computing capability (i.e., GFlops) and memory capacity (i.e., GPU global memory size). Since GPU memory is a major constrained resource, S^3DNN leverages a classical bin-packing algorithm to dispatch videos by considering GPU memory capacity as the bin size. The S^3 scheduler is then in charge of scheduling the DNNs (virtual contexts) at the granularity of GPU kernel. In order to meet real-time constraints, S^3 scheduler uses a kernel-level least-slack-first (LSF) scheduling policy, which prioritizes kernels considering both their deadlines and subsequent kernels belonging to the same DNN instance. Note that in our current implementation, S^3DNN uses YOLO (Redmon et al., 2016) as the DNN engine which represents the fundamental DNN processing framework. However, as a general middleware solution, S^3DNN can be easily plugged into other existing DNN frameworks, requiring minor rewriting work.

Before diving into the detailed design of each component, we present an abstract definition for the system. Namely, our system is constructed from a set of frame processing requests, $\tau = \tau_1, ..., \tau_n$, in which τ_i is eventually assigned to a DNN instance to be processed (this DNN instance can be shared by multiple τ_i). For the sake of simplicity, we assume each layer contains only a

single GPU kernel. If multiple kernels are used in a layer, we can combine them into a single GPU execution to achieve the same goal. We use this model throughout the chapter as a platform for any real-time analysis. In the case of data fusion, the DNN structure and algorithms function the same way, albeit with a larger data input.

4.2.2 System-level Data Fusion

As suggested by Insight 2, performing system-level data fusion upon DNN workloads may effectively improve performance in terms of throughput. Thus, S^3DNN implements a governor in the frontend-backend framework to selectively fuse incoming video frames at a system-wide level, by considering both throughput improvements and potential response time increases as executing fused frames may lengthen the execution time of individual frames. The goal is to conduct data fusion in a way such that real-time constraint is satisfied while throughput can be maximized.

Data fusion is possible due to an exploitation of matrix operations inside a DNN. For a normal DNN, most operations are matrix-based, following a format similar to the following:

$$R_{M*N} * F_{N*K} = M_{M*K}, \tag{4.1}$$

in which F is called filter, while R is the input for that layer and M is the output. Fusing α matrices together will turn Eq. 4.1 into:

$$R_{(\alpha*M)*N} * F_{N*K} = M_{(\alpha*M)*K}. \tag{4.2}$$

We consider the problem of efficiently fusing n videos with at most m different FPS targets into q DNN instances, each of which corresponds to a CUDA stream, where $m \leq q \leq n$. Each video is composed of a series of continuous frames that are processed sequentially. We define (e_k, d_k) to characterize each job in a DNN instance S_k, where e_k denotes the job's execution time on a GPU and d_i denotes the job's deadline (i.e., $1/f_i$ of the corresponding video where f_i denotes its target FPS). The design goal is to map τ to S. Before describing our developed data fusion

algorithm, there are several constraints we will apply. First of all, data fusion shall be performed only when the system observes multiple videos simultaneously. In other words, we never wait for more video streams to arrive in order to enable data fusion. Clearly, waiting for multiple frames to arrive and then fusing them together will significantly cause real-time requirements of an already arrived video to be violated. Moreover, in order to simplify the design complexity, only videos with the same DNN configuration (i.e., layers, weights, thus with the same e_i) will be fused into one DNN instance since they exhibit the same intensity of computation.

In addition to performing data fusion, the governor also needs to ensure that the total utilization of each DNN instance after fusion, denoted by U_k, is no greater than 1, where $U_k = \sum_{i=1}^{n_k}(\frac{e_i(S_k)}{d_i(S_k)})$ and $e_i(S_k)$ ($d_i(S_k)$) denotes the execution time and deadline, respectively, of the i^{th} fused video in the k^{th} DNN instance; otherwise, frame deadlines may be frequently missed which negatively impacts the real-time correctness. The pseudo-code of this algorithm is given in Algorithm 2.

A very interesting observation drawn while designing Algorithm 2 is that the total utilization of the system will not necessarily be equal to the sum of the individual utilizations $\sum_{\tau_1}^{\tau_n}(U_k)$. This phenomenon is due to the inherent massive parallelism in GPUs, which can lead to the total execution time of two combined kernels to be less than the sum of the execution times of each kernel due to better resource utilization. In order to mitigate this problem, Algorithm 2 uses a history-based approach to figure out real-world combined execution times instead of relying solely on individually measured execution times.

Algorithm description. As seen in Algorithm 2, our data fusion algorithm takes the task set τ as input in the form of a list of m deadlines of n frames. It will then go through all combinations of τ (input frames) and S (the current DNN instances) in lines 3 and 4. As was mentioned earlier, two tests need to pass. First and foremost, the configuration should match. Thus, Algorithm 2 will check if the deadline of the current task matches the deadline of the first task in a DNN instance (line 5). After that, the algorithm checks to see if adding this new task will violate the utilization test (line 6). This is done through the $LookUpHistory()$ function. This function takes the existing

Algorithm 2 Data Fusion Algorithm

Require: τ[] of all n inputs
Require: Instances[] of all DNN instances
1: **function** FUSE(InputList,Instances)
2: S[]=$\emptyset$
3: **for** i in τ **do**
4: **for** j in S **do**
5: **if** i.d == j[0].d **then**
6: **if** LookUpHistory(j,i) < 1 **then**
7: j.insert(i); EXIT()
8: Instances = CreateDNNInstance(S.length, FuseData(S))

tasks in a DNN instance along with a new task as input. It will then probe the history table to see if any history regarding this combination of tasks exists. If not, it will simply add up the individual utilizations to calculate the upper bound. If no such DNN instance is found, the algorithm will create a new one (line 8).

Memory-constrained dispatching. After the governor is finished fusing incoming videos, it will dispatch the fused video streams to different GPUs. Since GPU memory capacity is the major resource bottleneck herein, we transform this problem into a classical bin-packing problem and apply existing methods for dispatching. Specifically, the GPU memory size can be viewed as the bin size, while the memory required by each fused video stream is viewed as the item size. The problem is to pack all the fused video streams into GPUs such that the number of required GPUs is minimized. We thus leverage the classical first-fit algorithm to resolve this problem, which has been proven effective in many cases (Augonnet, Thibault, Namyst, and Wacrenier, Augonnet et al.).

Optimizations in the governor. S^3DNN uses history-based prediction for each kernel launch by storing its execution time information together with system information (input size, other kernels, etc.), as well as a hash value representing this kernel, in a lookup table. In addition to our previously mentioned observation, this approach is also based on a realistic assumption that the kernels used in a given DNN are fixed, and the critical variable that impacts execution time is the

input data size as well as other kernels. This lookup table is saved to a file and loaded into memory when S^3DNN boots up.

Our implementation of S^3DNN optimizes memory usage. Specifically, instead of constructing a copy of a separate DNN instance (which includes the model and input/output data) for each input video, S^3DNN optimizes memory usage by sharing model data among multiple DNN instances in GPU global memory. In practice, since all input video streams may use the same DNN configuration for specific purposes (e.g., object recognition with the same accuracy goal), S^3DNN would only need to store one copy of the model data in each GPU's device memory, shared by multiple DNN instances.

4.2.3 Supervised Streaming and Scheduling

As motivated by Insights 3 and 4 discussed in Sec. 4.1.3, S^3DNN seeks to optimize the execution of fused video streams assigned to each GPU through supervised streaming and scheduling, with the goal of maximizing concurrency (thus throughput and GPU utilization) and real-time performance. With the help of the "per-thread" streaming option in CUDA 7 or newer versions, streaming-enabled concurrency becomes even easier to achieve, because the hardware scheduler inside the GPU computing engine takes care of the throughput optimization. Unfortunately, when kernels are submitted to GPU hardware by the driver, they do not have any priorities, causing the kernels to be executed in a FIFO order. This may jeopardize another important real-time performance indicator for DNN-based real-time object recognition workloads: deadline meeting ratio. Motivated by this, S^3DNN develops a GPU scheduling algorithm incorporating several novel ideas to simultaneously achieve these two (sometimes) conflicting goals. Specifically, this scheduler extends a classical real-time scheduler least-slack-first (LSF) to improve real-time performance, combined with supervised streaming via a lookahead approach for maximizing concurrency benefits. We choose to use kernel-level LSF because LSF, as a dynamic priority scheduler, can make kernels from different DNN instances be assigned different priorities and execute in an

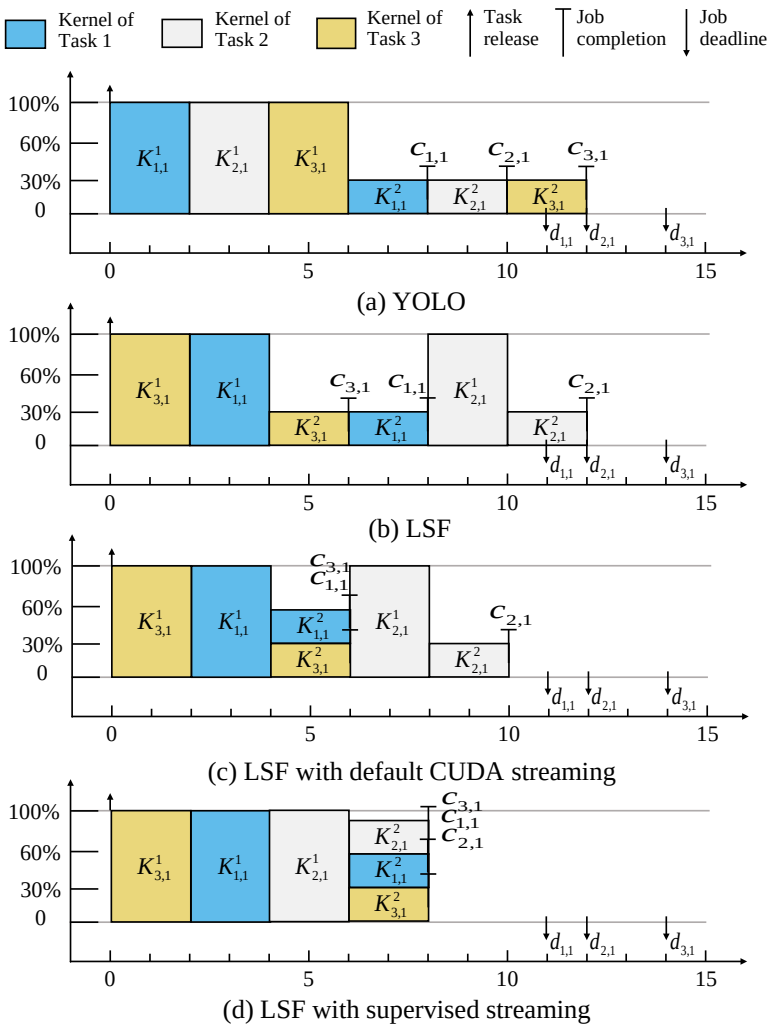


Figure 4.5: Comparison of four scheduling policies.

interleaving manner. The slack of a kernel at time *t* is defined to be the deadline of the corresponding job of the DNN task minus t, and then minus the total remaining amount of execution of this task's kernels in the current job period. Static priority schedulers such as RM and other dynamic priority schedulers such as EDF will always assign the same priority to kernels belonging to the same DNN instance, preventing concurrency to be implemented efficiently. Regarding the concern of runtime overhead, LSF is similar to EDF in this case. This is because each kernel execution on GPU is non-preemptive, implying that the slack value of a DNN task (used to decide the priority of a pending kernel) only needs to be updated when any of its kernels complete execution on GPU. We first use the following example to illustrate the fundamental ideas behind S^3DNN's scheduler.

Algorithm 3 Supervised streaming and scheduling algorithm

Require: Q ▷ Queue of kernels to be scheduled
Require: G ▷ Group of kernels that can execute concurrently
Require: C ▷ Critical queue that will be submitted to GPU

```
1:  function ENQUEUE(Q[],k)
2:      for q in Q do
3:          UpdateSlacks(Q)
4:          if Slack(k) > Slack(q) then
5:              Q.InsertBefore(k, q)
6:  function DEQUEUE(Q[])
7:      UpdateSlacks(Q)
8:      if G ≠ ∅ then
9:          go to assign
10:     h ← HeadOf(Q)
11:     G.insert(h)
12:     if tbRatio(h) < 1 then
13:         for t in Q-h do
14:             if tbRatio(t) < 1 then G.insert(t)
15:             if tbRatio(G) > 1 then break
16:     else go to assign
17:     if tbRatio(G) < 1 then
18:         h' ← HeadOf(Q)
19:         p ← LookAhead(h')
20:         if tbRatio(p) < 1 then
21:             G.reserve(p); C.insert(h')
22:             go to submit
23:         else go to assign
24:     C ← G; G ← ∅
25:     Submit(C); Q.Remove(C)
```

Consider three DNN tasks K_1, K_2, K_3, each of which contains two layers. The first layer utilizes 100% of the thread block resource in a GPU, and the second layer utilizes 30% of the thread block resource (following the same pattern observed in Sec. 4.1). K_1, K_2, and K_3 have a deadline of 12, 14, and 11 time units respectively. Fig. 4.5 illustrates four possible schedules under four different scheduling and streaming methods, where the y-axis represents the percentage of the thread block resource required by a kernel.

Fig. 4.5(a) shows the FIFO schedule for isolated processes which is the default behavior used in the YOLO framework, which causes a deadline miss for K_3 due to serialized execution and deadline-oblivious prioritization under FIFO. Fig. 4.5(b) shows the schedule under a kernel-level non-preemptive LSF scheduling algorithm, which prioritizes kernels according to least-slack-first. A kernel $K_{i,j}^k$'s slack is defined to be $r_{i,j}^1 + d_{i,j} - r_{i,j}^k - F(r_{i,j}^k)$, where $r_{i,j}^k$ denotes the release time of the k^{th} kernel of the j^{th} job of the i^{th} DNN instance, and $F(r_{i,j}^k)$ denotes the amount of computation completed by kernel $K_{i,j}^k$ at time $r_{i,j}^k$. Intuitively, the slack denotes the number of time units a DNN task can use to complete the remaining computation of the corresponding released kernel. Note that $K_{i,j}^k$ is released at the time when $K_{i,j}^{k-1}$ (if any) completes. Thus, under the kernel-level LSF, a kernel's priority is defined using LSF when it is released. As seen in Fig. 4.5(b), the kernel-level LSF is able to meet all tasks' deadlines, yielding an end-to-end response time of 12 time units. Note that LSF will not incur much overhead at runtime since priority definition is determined only at kernel boundaries, and the maximum number of kernels released in the scheduling queue equals to the number of CUDA streams which is often small, for example, less than 6 in our evaluation.

Although applying kernel-level LSF improves real-time performance, it does not benefit from concurrency. Thus, we illustrate in Fig. 4.5(c) a schedule under the kernel-level LSF scheduler combined with default CUDA streaming (i.e., put each DNN task into a separate stream and run the three streams concurrently). As seen in the figure, since K_1^2 and K_3^2 only use a total amount of 60% of the thread block resource, these two kernels can execute concurrently at time 4. Doing so further improves the end-to-end response time to be 10 time units.

An interesting observation obtained from Fig. 4.5(c) is that if K_1^2 and K_3^2 can be supervised to wait for the release of K_2^2, then these three kernels can actually be executed concurrently, which results in a further response time performance improvement as well as a better overall GPU utilization and throughput, as illustrated in Fig. 4.5(d). As seen in this figure, a supervised delay happens at time 4, where the scheduler chooses to run a lower-priority task K_2^1 first, thus allowing those three kernels to run together. This key observation motivates our design of S^3DNN's GPU

scheduler. Since DNN tasks fundamentally exhibit a staged computation pattern where later layers often require fewer resources (i.e., Insight 1), it is more likely that later kernels belonging to different DNN tasks can be executed concurrently. However, since a kernel is released and pushed into the scheduling queue only if its predecessor kernel completes, at each scheduling instance (i.e., when a currently running kernel completes on GPU), our scheduler takes a "lookahead" approach to find whether the highest-priority kernel in the scheduling queue can run concurrently with any later released kernels, as well as any kernels already waiting in the scheduling queue. This idea is illustrated by the example shown in Fig. 4.5(d), where at time 4, the scheduler lookaheads and finds out that the highest-priority kernel in the queue at time 6, which is K_3^2, can concurrently run with a later released kernel K_2^2 and an already released kernel K_1^2.

Algorithm description. Motivated by the above-discussed ideas, we now present our developed GPU scheduler for S^3DNN. We use a metric, $tbRatio$, to measure the proportion of the demanded thread blocks by a kernel to the total number of thread blocks provided by the GPU hardware (often constrained by either the hardware architecture or register/shared memory size). For example, GPUs with compute capability 2.x can support up to eight blocks per SM, if register/shared memory size is not the bottleneck. The pseudo-code of the algorithm is given in Algorithm 3. There are two major functions defined in Algorithm 3: Enqueue (Line 1) and Dequeue (Line 6). The algorithm needs three input data structures, including a scheduling queue (denoted by Q), any subset of kernels that can execute concurrently (denoted by G), and the subset of kernels with the highest priority in the scheduling queue (denoted by C). Function Enqueue is invoked at kernel arrivals and completions, which sorts the released kernels in Q using LSF (Lines 2-5). It first updates the remaining slacks for each kernel (line 3), and then inserts each newly arrived kernel k into Q according to LSF (Lines 4-5). Instead of invoking Dequeue immediately after a kernel's (e.g., k_0's) completion, S^3DNN delays this invocation a little bit until k_0's successor kernel is pushed into the queue. It first updates slacks for kernels in the scheduling queue (Line 7), and then checks if G is empty (Line 8). If G is not empty, then the scheduler directly submits kernels within G

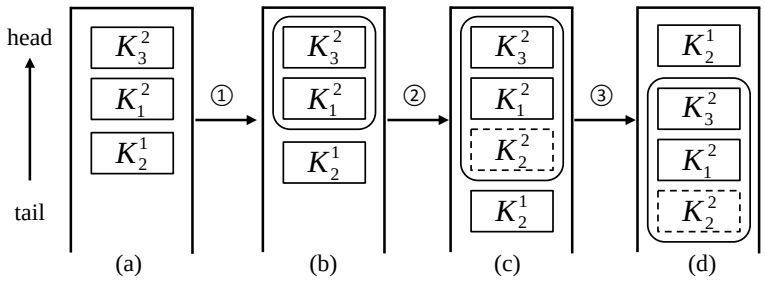


Figure 4.6: Intuitive illustration of Algorithm 3 using the example given in Fig. 4.5 (d).

for execution (Line 9). Next, the kernel h at the head of the scheduling queue is pushed into G (Lines 10-11). Then the scheduler checks whether h can fully occupy the GPU by calculating its $tbRatio$. If $tbRatio$ of h is smaller than 1 (Line 12), indicating it may be concurrently executed with other kernels, then the scheduler will seek to put more kernels in Q whose $tbRatio$ is also less than 1 (Lines 13-15); else h is directly submitted to GPU device for execution (Line 16). After all potential small kernels in Q are merged into G, the scheduler checks whether the $tbRatio$ of G is still less than 1 (Line 17). If so, the scheduler looks ahead the successor kernels of the ones residing in Q in the order of priorities, in order to identify any such kernels that have not been released but with $tbRatio < 1$ (Lines 18-22). The looking ahead operation can be achieved because S^3DNN builds dependency graphs (linear in YOLO) for all kernels at DNN instance construction. The purpose of doing this lookahead method is to check whether it is possible to have a maximum set of kernels that can run concurrently. If such a successor kernel exists, then the scheduler will supervise the set of kernels in G to wait for the release of this successor kernel. Considering the potential overhead caused by this lookahead method in practice, we limit the lookahead degree to one, which implies that the scheduler will only check the subsequent kernel released by the next highest-priority kernel in the scheduling queue (not counting the kernels already placed in G since they need to complete first in order to release subsequent kernels). Finally, kernels placed in G will be sent to C, which will be further submitted to GPU for execution (Lines 24-25).

Algorithm illustration. We use an abstracted workflow diagram shown in Fig. 4.6 to illustrate how our proposed scheduler schedules the example DNN task set given in Fig. 4.5(d). At time 0,

Table 4.2: Configuration of video numbers and FPS

Config	light			medium			heavy	
FPS	3	4	5	3	4	5	3	4
10 FPS	n/a	n/a	n/a	n/a	n/a	n/a	1	2
15 FPS	n/a	n/a	n/a	1	2	2	2	2
20 FPS	1	2	2	2	2	3	n/a	n/a
25 FPS	2	2	3	n/a	n/a	n/a	n/a	n/a

three DNN tasks are released simultaneously. According to their slack times, the priority queue is sorted in the order of $K^1_{3,1}$, $K^1_{1,1}$ and $K^1_{2,1}$. The scheduler will thus dequeue and schedule $K^1_{3,1}$ first, and then $K^1_{1,1}$ since there is no chance to run either kernel concurrently with another kernel. At time 6, three kernels wait in the scheduling queue in the order of $\{K^2_{3,1}$, $K^1_{2,1}$, and $K^2_{1,1}\}$ (Fig. 4.6(a)). Since the highest-priority kernel $K^2_{3,1}$ has a thread block utilization of 30%, the scheduler will scan the other kernels in the scheduling queue to increase concurrency. In this case, it finds that $K^2_{1,1}$ can be grouped with $K^2_{3,1}$ to execute concurrently (Fig. 4.6(b)). Then, since the thread block utilization of these two grouped kernels is 60%, still less than 1, the scheduler seeks to lookahead to check whether there is any later-released kernel that can execute with this group together. With a lookahead degree of 1, the scheduler will only check once whether the successor kernel of the the second-highest-priority kernel in the queue, which is $K^1_{2,1}$, can be grouped together. In this case, the total thread block utilization of the three kernels $K^2_{3,1}$, $K^2_{1,1}$, and $K^2_{2,1}$ is 90%, thus eligible to run concurrently. Thus, the scheduler will group these three kernels together and will reverse the priority ordering of this group of kernels with $K^1_{2,1}$ which is originally the second-highest-priority kernel waiting in the queue, to maximize concurrency. The scheduler has to schedule $K^1_{2,1}$ first because $K^2_{2,1}$ will not be released untile $K^1_{2,1}$ completes.

In summary, S^3DNN's scheduler is designed to improve real-time performance through adopting a deadline-aware LSF algorithm while simultaneously maximizing concurrency through supervised streaming.

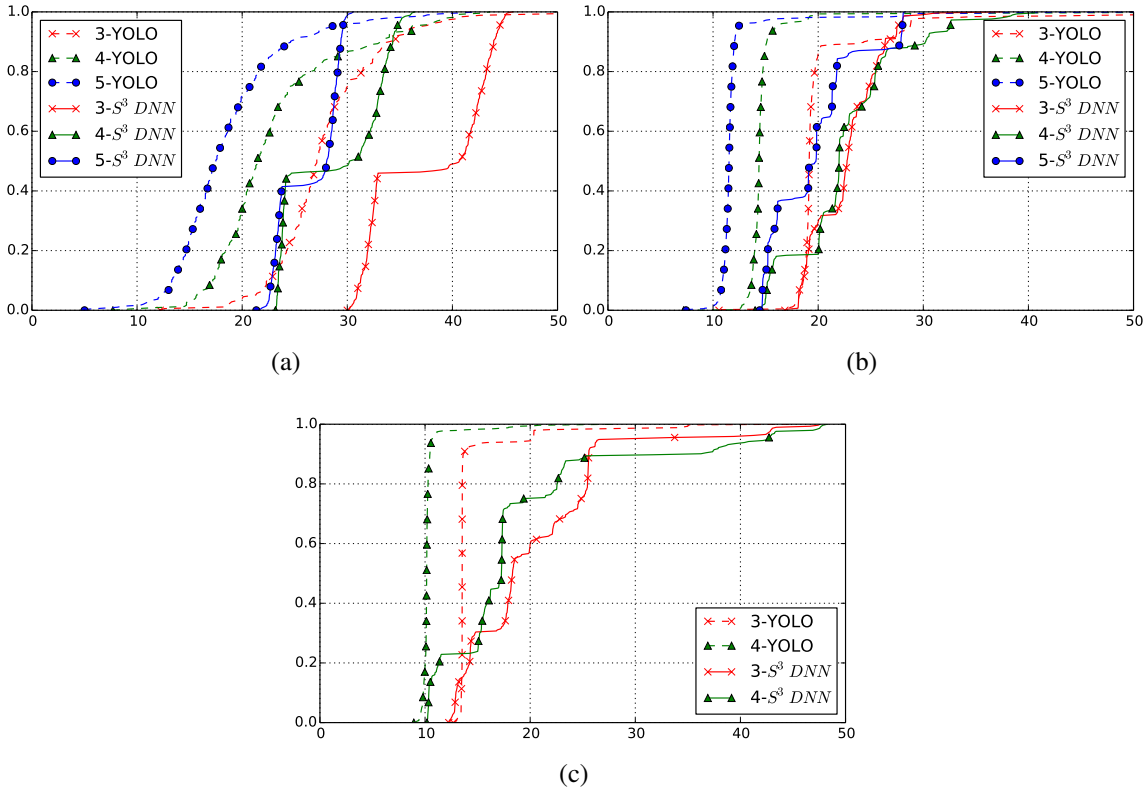


Figure 4.7: CDF of FPS under (a) light (b) medium (c) customized DNN configurations. "3-yolo" ("3-S^3DNN") represents the FPS performance under YOLO (S^3DNN) when there are three input vidoes.

4.3 Evaluation

4.3.1 Experiment Setup

We conduct our experiments in a system consisted of an NVIDIA Quadro 6000 GPU, which is based on the Fermi micro-architecture and features 480 cores and 6 GB of GDDR5 memory, and an Intel Core i7-4790k CPU and 16 GB of RAM. We use Ubuntu 14.04 based on Linux 3.5.7 as the underlying operating system. We compare S^3DNN to the YOLO framework as the baseline which is seeing widespread use in both academia and industrial systems for processing DNN workloads (Redmon et al., 2016). In this section, we will evaluate both real-time and throughput performance in terms of end-to-end response time and FPS under multi-tasking scenarios. We first

conduct extensive experiments using video streams stored on the disk drive, and then a case study using live video streams captured in real-time by using on-dash mounted cameras. The frame frequency of a video as well as the deadline of each frame is determined by the videos' FPS. We use videos with 4 different FPS: 10, 15, 20, and 25. We evaluate three DNN configurations: one is configured with a shallow network, small weight file, and fast kernels, representing a light configuration; the second one is configured with a deep network, large weight file, and fast kernels, representing a medium configuration; the third one is configured with a deep network, large weight file, and slow kernels, representing a heavy configuration. These three configurations cover the two major aspects that affect the execution: execution time of each layer (fast or slow kernels) and number of layers (shallow or deep DNN layout).

4.3.2 Real-time performance

We first evaluate the real-time performance of S^3DNN by measuring the cumulative distribution function (CDF) of FPS under nine scenarios combining three DNN configurations with three cases of different input videos (3, 4, and 5 videos, respectively). For each experiment, we use a mixed set of video streams with different FPS, as shown in Table 4.2. The second row indicates three workloads configurations. The first column uses FPS to indicate the deadlines of videos. Columns 2 to 9 show the number of concurrent videos and their composition. For example, the second column shows that there are three concurrent incoming videos in total, in which one is 20 FPS and two are 25 FPS.

Fig. 4.7 shows the evaluation results, where the x-axis represents FPS and the y-axis represents CDF. We observe that when the number of videos is more than one, S^3DNN outperforms YOLO by a significant margin. For example, with three input videos and light DNN configuration, S^3DNN is able to achieve an FPS higher than 30 in almost all cases, while 80% of the frames have an FPS lower than 30 under YOLO. Moreover, in many cases, S^3DNN is able to provide a higher FPS than the original FPS of the input videos. This implies that under S^3DNN, most of

the video frames can be completed before their deadlines (note again that a frame's deadline is defined as its release time plus 1/FPS), thus meeting the real-time processing constraint. On the other hand, YOLO yields rather low FPS performance in many cases, particularly when the workloads are heavy and/or DNN configuration becomes heavy. Also note that as seen in Fig. 4.7(c), when the DNN configuration is heavy, we cannot get results from either YOLO or S^3DNN because the workload significantly over-utilizes the available hardware resource. Generally speaking, as is seen in the three insets of Fig. 4.7, with increased number of videos, and/or heavier DNN configurations, the overall performance under both YOLO and S^3DNN decreases. This is intuitive because more workloads will cause heavier contention on GPU. However, even under the scenario with the heaviest workload (i.e., heavy DNN configuration with 4 input videos as seen in Fig. 4.7(c)), S^3DNN is still able to provide reasonably good performance, where CDF of 15 FPS or above is greater than 70%; while YOLO yields a performance below 10 FPS for almost every frame in this case. Thus, our design of S^3DNN proves to be much more effective in delivering real-time performance through performing data fusion and supervised streaming and scheduling.

4.3.3 Overall Throughput

In this set of experiments, we mainly evaluate the throughput under S^3DNN compared to YOLO. For the three DNN configurations, we use multiple 25-FPS one-minute-long videos as input. Since we use offline videos as inputs, once one frame is processed, the next frame can be immediately processed afterwards. The throughput is shown in the left sub-figure of Fig. 4.8, where the x-axis is the number of videos processed simultaneously, the y-axis is the normalized throughput (i.e., the ratio of the throughput under S^3DNN divided by the throughput under YOLO). The four histograms shown under each case represent YOLO and S^3DNN under three different DNN configurations. We observe that S^3DNN outperforms YOLO when there are multiple input videos, fundamentally due to the fact that S^3DNN enables concurrency through supervised streaming. When there is only one video in the system, YOLO actually yields a slightly better performance

than S^3DNN. This is because of the runtime overhead introduced by S^3DNN, which is reasonably small (less than 4% for all three DNN configurations). Another interesting observation is that S^3DNN yields the biggest performance improvement when there are two or three videos in the system with a heavy DNN configuration. This is because (*i*) the heavy DNN configuration implies heavier workloads that may benefit more from concurrency due to supervised streaming and scheduling (as discussed earlier), and (*ii*) data fusion becomes particularly effective in such cases because it is possible to fuse two or three videos into a single DNN instance. Processing a single large matrix-based computation can be more efficient than separately processing several smaller matrix-based computations. For cases with four or five video streams, the throughput under S^3DNN decreases because four videos cannot be fused into a single DNN instance due to the fact that such data fusion would cause the processing time of the resulting fused frame to be definitely longer than the deadline of each individual frame.

The right sub-figure of Fig. 4.8 demonstrates the breakdown of the histogram representing five heavy workloads scenario. The histogram labeled "Fuse" ("S^3") indicates the throughput when S^3DNN only enables the functionality of data fusion (S^3 scheduler). We observe that data fusion brings the majority of throughput improvement (10%). S^3 scheduler still brings 6% throughput improvement though it is designed for meeting real-time constraints. The combination of both brings 12% improvement (as shown in the left sub-figure) which is not linearly added-up of two components' improvements. This is because when we use each standalone technique, the optimizations realized in the frontend-backend framework (i.e., model sharing, GPU context consolidation) benefit both scenarios.

4.3.4 Assessing the Supervised Streaming and Scheduling Module

As we believe the supervised streaming and scheduling module contributes to both real-time performance and throughput, we conducted a set of experiments specifically evaluating the efficacy

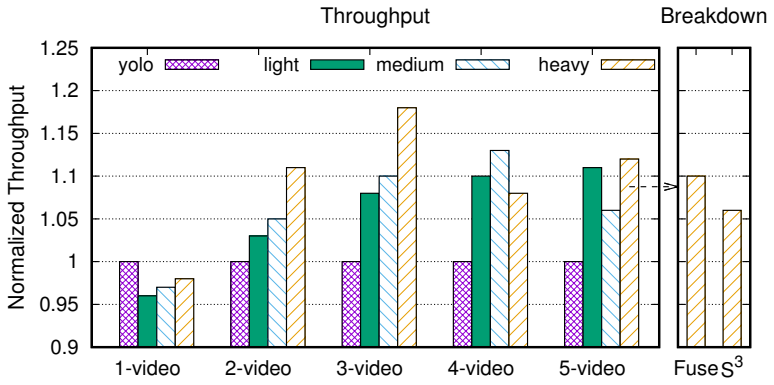


Figure 4.8: Normalized throughput under light, medium, and heavy workloads using a variant number of videos.

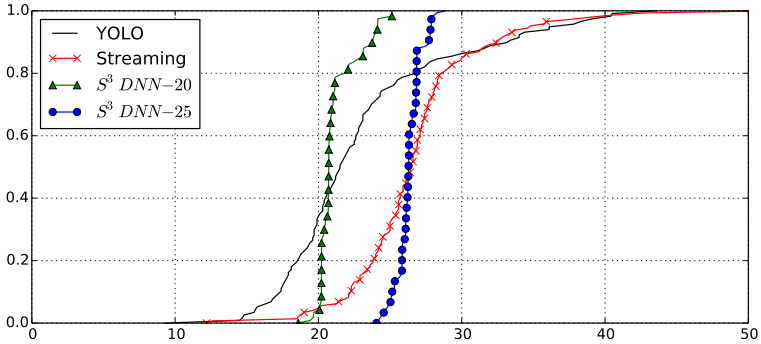


Figure 4.9: Efficacy with respect to FPS under S^3 compared to isolated run and default CUDA streaming

of this module. As discussed in Sec. 4.2.3, we compare this module against two other methods, including YOLO and concurrency-enabled CUDA streaming (i.e., binding multiple DNN instances to separate CUDA streams and running them concurrently by the GPU hardware scheduler). In these experiments, we use the light DNN configuration and four input videos, two 25-FPS videos and two 20-FPS videos. We use the CDF of FPS as the evaluation metric.

Fig. 4.9 shows the experimental results, where the performance under S^3DNN is separated into two curves, corresponding to the two 20-FPS videos and the two 25-FPS videos, respectively. For the other two approaches, we do not separate the results according to FPS. This is because both these approaches are application-oblivious. Thus, the resulting performance for each of the four videos is almost identical, even though they have different FPS. For depiction clarity, we just

draw one curve that represents the overall performance for processing all four videos under both approaches. Note that since S^3DNN schedules tasks considering their deadlines (thus FPS), the resulting performance under each FPS category becomes distinguishable.

We observe in Fig. 4.9 that both CUDA streaming and our supervised streaming methods improve the performance compared to YOLO. Compared to CUDA streaming, the FPS performance under our supervised approach is clearly more predictable and consistent. For example, under CUDA streaming, over 30% frames of the two 25-FPS videos miss their deadlines (i.e., with a $<$ 25 FPS); while under S^3DNN, only 6% frames of the two 25-FPS videos miss their deadlines. Note that S^3DNN yields a slightly worse overall performance than CUDA streaming, because prioritizing kernels in the scheduling queue and performing supervised synchronization introduces runtime overheads, particularly when the queue has a large number of kernels.

4.3.5 Multi-GPU scenarios.

Since S^3DNN can also be applied in a multi-GPU environment, we have conducted experiments to evaluate its performance in such scenarios. The evaluation platform is a heterogeneous multi-GPU system consisting of an NVIDIA GTX 480 device and an NVIDIA Quadro 6000 device, and an Intel i7 multi-core CPU. The NVIDIA GTX480 device features higher GFLOPS but less memory capacity compared to NVIDIA Quadro 6000. As briefly discussed in Sec. 4.2.2, for multi-GPU systems, the Governor in S^3DNN applies an efficient bin-packing heuristic to assign fused video streams onto GPUs according to each GPU's memory constraints.

The baseline compared in this experiment is the best performance among various possible partitioning of YOLO instances onto two GPU devices. For example, when the number of videos is five and light DNN configuration is used, the baseline approach will assign three videos to GTX 480 and another two to Quadro 6000, which yields the best throughput among all partitioning possibilities.

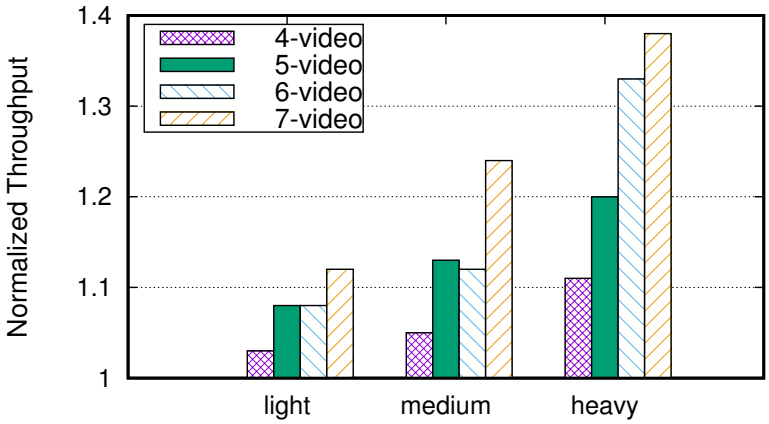


Figure 4.10: Performance under multi-GPU scenarios.

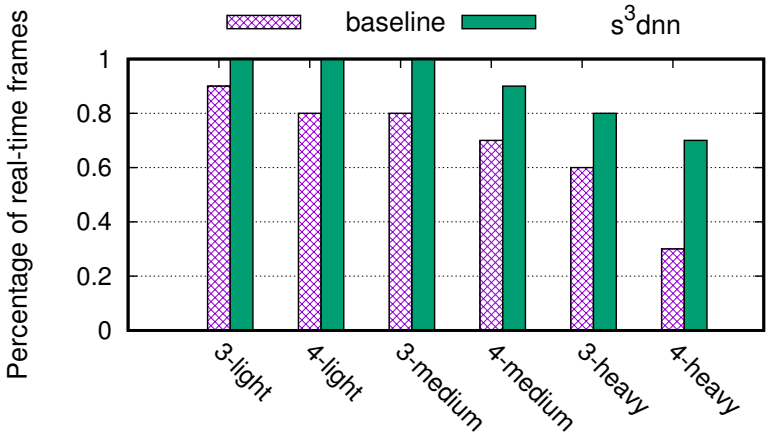


Figure 4.11: Percentage of frames that meet their deadlines.

Fig. 4.10 shows the throughput performance of S^3DNN compared to YOLO under different DNN configurations when processing different number of input videos. As seen in the figure, S^3DNN significantly improves the throughput performance under all scenarios. The largest improvement (almost 40% improvement) occurs under the scenario with heavy DNN configuration and seven input videos. This is because in this case, the baseline solution assigns two videos on fast (high GFLOPS) but small (GPU memory size) GPU (i.e., GTX 480) due to memory constrains, leaving five video on slow but large GPU (i.e., Quadro 6000). This unbalanced assignment causes throughput loss. On the other hand, due to data fusion, S^3DNN can achieve a much more balanced partitioning, thus yielding a better throughput performance.

4.3.6 Online Webcam-based Object Recognition

Besides evaluating S^3DNN for offline video-based real-time object recognition, we conducted a case study evaluating the efficacy of S^3DNN under an online scenario, where online webcams continuously capture real-time video frames that require real-time object recognition processing. The major difference between using online webcams and offline videos is the following: for offline videos, the release time of each frame is rather flexible, i.e., whenever a frame completes, the next frame can be immediately released and fed to the system; while for online webcam-captured video streams, the release time of the frame is fixed, as defined by the FPS of the webcam. Moreover, webcams often use a buffer to cache captured frames, with the advantage of hiding I/O latency. However, if the processing speed is slow, then the buffer will overflow and harm the real-time performance.

In this set of experiments, we use up to 4 online webcams, each of which is configured to be 15 or 20 FPS. We compare against YOLO, where each webcam is bound to a YOLO instance. Fig. 4.11 shows the results in terms of the percentage of the frames that meet their deadlines. The x-axis shows the evaluated scenarios, e.g., "3-light" denotes the scenario with three webcams and light DNN configuration. We observe that S^3DNN clearly outperforms YOLO in all scenarios, particularly when the system needs to process more workloads due to an increased number of webcams and/or heavier DNN configurations. For example, in the "4-heavy" case, baseline can merely process 30% of the frames in real-time while S^3DNN can achieve a schedulability of nearly 70%. Thus, S^3DNN is also effective in processing online video streams with enhanced real-time performance.

4.4 Summary

In this chapter, we present S^3DNN–a systemic solution that optimizes the execution of DNN workloads on GPU in a real-time multi-tasking environment. Experimental results show that

S^3DNN significantly outperforms state-of-the-art GPU-accelerated DNN processing frameworks in a real-time multi-tasking environment.

CHAPTER 5

TIMING-PREDICTABLE ENERGY OPTIMIZATION FOR DEEP NEURAL NETWORKS [1]

5.1 Motivation

In this section, we present two fundamental observations that have motivated our design and implementation of an efficient system solution specifically tailored for DNNs.

5.1.1 DNN-specific Energy Usage Patterns

We have performed an extensive set of experiments seeking to investigate the energy usage patterns in a neural network. We setup the default configuration of Alexnet on a Jetson TX2 as a constant factor and measure the energy consumption and the compute latency under different configurations.

The effect of GPU and CPU frequency on energy consumption for general workloads have been extensively explored before (Abe et al., 2014; Kim et al., 2015; Imes et al., 2015; Santriaji and Hoffmann, 2016; Zhang and Hoffmann, 2016). However, on an NVIDIA Jetson TX2, we found no clear linear pattern between lowering/boosting the frequency and decreasing/increasing the energy usage when running a DNN instance as a single entity. Rather interestingly, memory energy usage decreases exponentially with higher frequencies. This behavior is more related to the inherent structure of memory. Thus, making memory operations shorter by increasing the memory frequency would result in a better energy efficiency.

In the case of GPU, the inherent parallelism causes the execution time of Alexnet to exponentially decrease when the frequency is increased, outpacing the increased power usage that results from the higher frequency. However, there is also not a clear trend when GPU frequencies are set

[1] ©2018 IEEE. Reprinted, with permission, from Husheng Zhou, Soroush Bateni, and Cong Liu. "PredJoule: A Timing-Predictable Energy Optimization Framework for Deep Neural Networks", In Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS18). This paper is not published by IEEE at the time when writing this dissertation.

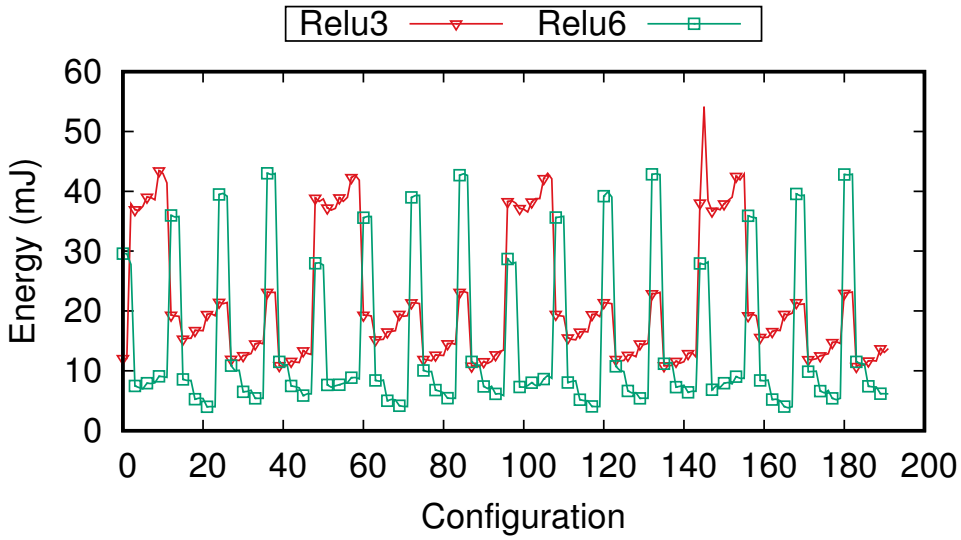


Figure 5.1: Energy usage of Relu3 and Relu6 under different DVFS configurations.

to be higher than 1000MHz. For example, it is possible for a configuration with GPU frequency of 1000MHz to be more energy efficient compared to one with a 1300MHz frequency.

We next investigate the per-layer energy usage patterns in neural networks. For demonstration purposes, we only show 2 different layers among all 24 default layers in Alexnet in Fig. 5.1. For this experiment, we combine all the frequency changes but only present a subset of 192 configurations in Fig. 5.1 for clarity. We record the average energy usage on an NVIDIA Jetson TX2. As is depicted in the figure, each layer demonstrates a different response to the configuration change. Interestingly, the same type of layer, Relu, behaves vastly different (almost opposite) when positioned at different depths. This stark difference is due to the fact that by the time Alexnet reaches Relu6, the data has been shrunken to a point that Relu6 is dominated by computation rather than memory. The same behavior is observed among all layers, with almost no consensus throughout the neural network on an optimal configuration for energy usage. This trend is observed even for the entire 14111 configurations tested.

Observation 1: If we consider DNNs as a blackbox, choosing the best configuration would not be as simple as solving a linear equation (as for other more straightforward workloads (Kim et al., 2015)) since our measurements show that Alexnet behaves optimally only by changing the

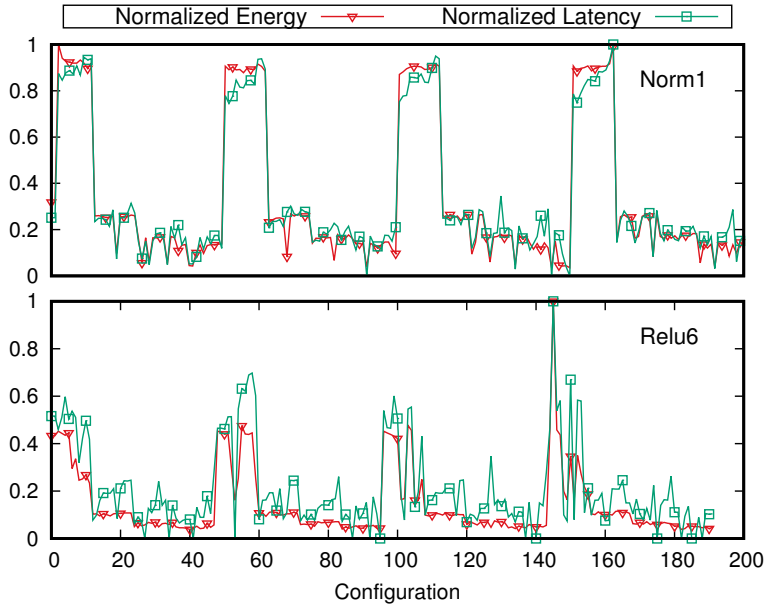


Figure 5.2: The trailing effect for two layers. Layer 4 has a pronounced trailing effect while layer 18 does not.

DVFS configuration for each layer. In fact, a layer's optimum configuration can be the absolute worst for other layers, making the process of finding an optimal configuration rather challenging and interesting for neural networks. Moreover, we find that layer type alone cannot be a deciding factor in finding the optimal configuration.

5.1.2 Energy-Performance Relationship

While choosing an optimum energy configuration for neural networks is an interesting challenge on its own, addressing this problem without considering latency can cause safety-critical timing issues. An optimal energy configuration for a layer can result in unreasonably long processing times.

Thus, we have also considered layer-based performance in terms of latency. We would like to investigate how a potential solution might be able to balance between energy usage and performance such that stringent latency requirements are met while energy gets minimized. Specifically, we are interested in understanding how each layer might respond to system configuration change

in terms of latency and energy efficiency. Fig. 5.2 shows the same setup used for Fig. 5.1. In this case, we include the latency metric for the first normalization layer Norm1 and the Relu6 layer in Fig. 5.2. Values of energy and latency are normalized by dividing each value by the maximum of their respective measurement.

Naturally, when the system jumps from one configuration to another, there will be changes to both latency and power. The critical point is to understand how would the energy consumption respond to a decrease or increase in latency. As seen in Fig. 5.2, for Norm1, a decrease in latency due to configuration changes will almost always result in a decrease in energy consumption and vice versa. On the other hand, for a layer like Relu6, this trend is not clear. In other words, we could be certain to a point that increasing the speed for Norm1 will result in better energy efficiency. However, we cannot be certain that such a benefit exists for Relu6. For Relu6, it is possible that a slight increase in speed can cause significant energy loss.

We call the trend present in Fig. 5.2 the trailing effect because energy consumption trails the latency. In Sec. 5.2, we present an idea that can separate the layers based on this trailing effect. Moreover, we will show the degree of which the trailing effect exists for various neural networks and layers in Sec. 5.3.

Observation 2: We have investigated the impact of DVFS configurations on latency performance with relation to energy, observing that a trailing pattern often exists between energy and performance. This trailing relationship motivates us to explore potential solutions that may exploit per-layer characteristics for more efficiently saving energy while maintaining timing requirements. For instance, compared to Relu6, layer Norm1 could be a much better candidate if we need a certain speedup while executing the network in order to meet the corresponding deadline.

5.2 Design

In this section, we outline the overall design of PredJoule, as illustrated in Fig. 5.3. To exploit per-layer characteristics in terms of latency and energy usage patterns, we introduce a concept

of Uncertainty for each layer. Uncertainty indicates how uncertain the system should be about a layer's energy to performance relationship (as discussed in Sec. 5.1). Intuitively, the lower the value of Uncertainty, the better the chance of achieving the optimal power/performance ratio for that layer.

Consequently, the system should aggressively change the DVFS configuration to a faster one for layers that have low Uncertainty. Moreover, the controller should explore more configurations for layers with a high Uncertainty because the behavior of these layers is inherently unpredictable.

To maintain timing correctness, PredJoule introduces a progress tracker that provides an effective system status and interference control in the form of an execution deficit ε. The progress tracker then conveys ε to a controller that is responsible for making configuration changes at runtime. The controller is able to react to ε via configuration adjustments. However, PredJoule imposes a strict power-consumption tariff on the decision-making process of the controller. This is achieved by inputting the next layer's Uncertainty value into the controller. By smartly integrating the consideration of both execution deficit ε and the next layer's Uncertainty, PredJoule is able to find the optimal configuration for each layer that yields controlled timing correctness with minimized energy consumption. Finally, we address issues of error-prone partial sorting and timing guarantee with a history module. If the history is reliable enough, our method will use history instead to adjust the total execution time accurately. Note that the configuration adjustment decision is made only at each layer's boundary (i.e., at the completion time of each layer's execution).

5.2.1 Uncertainty

To capture the layer-based uniqueness in power and performance usage, we introduce a new concept: Uncertainty. As discussed in Sec. 5.1, a unique characteristic of some DNN layers was the trailing effect. Layers that exhibit this effect can be good candidates for more aggressive configuration adjustments since the performance and energy behaviors can be easily predicted. Intuitively, Uncertainty reflects the degree of certainty the system has on predicting how a layer's energy usage

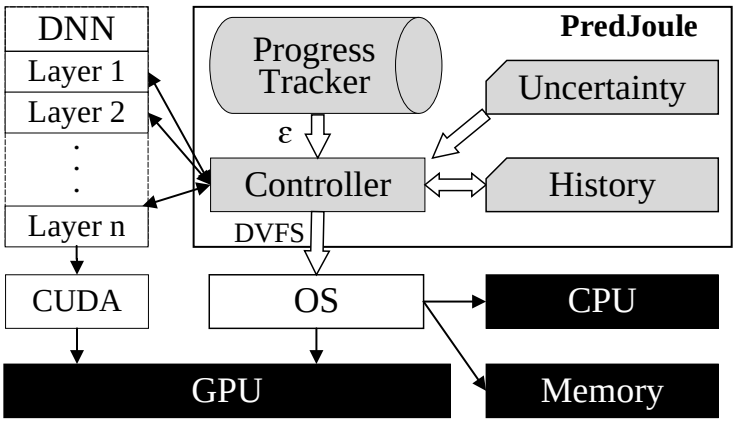


Figure 5.3: Design overview of PredJoule.

would respond to the performance change due to configuration adjustments. A lower Uncertainty value implies a higher certainty.

Before formally defining Uncertainty, we first define normalized energy and latency performance. We normalize the latency and energy usage values by dividing each value by the maximum:

Definition 1: Normalized energy and latency

$$Norm(P_{c\in C}) = \frac{P_c}{\max_{c\in C} P_c}, \tag{5.1}$$

$$Norm(L_{c\in C}) = \frac{L_c}{\max_{c\in C} L_c}, \tag{5.2}$$

in which $Norm(P_{c\in C})$ and $Norm(L_{c\in C})$ are the normalized energy and performance values under a configuration c, and C represents the entire configuration space.

Definition 2: Configuration Uncertainty (CU). For running a layer under a configuration $c \in C$, we define CU of the layer as:

$$CU_{c\in C} = \frac{Norm(P_c)}{Norm(L_c)}. \tag{5.3}$$

Since CU is on a per-layer and per-configuration basis, storing and processing it for hundreds of layers under thousands of configurations is expensive (if not impossible). Thus, we collect and

Table 5.1: Uncertainty for an example DNN configuration.

Conv1	Relu1	Norm1	Pool1	Conv2	FC	Softmax
0.8	10.4	854.0	5.4	1.7	56.7	125493.0

summarize CU for each layer, and reduce the problem complexity by $|C|$ (size of configuration space). We utilize the max function to define the Uncertainty:

Definition 3: Uncertainty. Given the set of configurations C for each layer, we define the Uncertainty as:

$$U = max_{c \in C} CU_c. \tag{5.4}$$

According to the above definition, the Uncertainty of a layer indicates the maximum gap between the normalized energy consumption and the normalized latency. If the energy consumption is high (close to 1) and latency is low (close to 0), Uncertainty would become extremely large. This large value can indicate to the system that a low latency (implying a fast configuration) can yield a high energy consumption. Thus, the system needs to be cautious since it is uncertain about whether or not a boost in performance could cause a dramatic sacrifice in energy for that layer. On the other hand, for the case where Uncertainty is low, the system knows that an increase in latency performance would lead to a smaller energy consumption for the corresponding layer and configuration. Thus, the system can be more certain on optimizing the latency and energy using this layer.

As an example, imagine a system with 7 computing layers. The Uncertainty of each layer along with their type is depicted in Table 5.1. We note that real-world neural networks are usually much deeper, yet Table 5.1 contains almost all the crucial layers in a compact configuration. As is seen in the table, the first Convolutional layer abbreviated as $Conv1$ has an Uncertainty that is low. This is due to the type of the layer. Moreover, this layer is at a shallow level of the DNN, which makes the trailing effect for this type of layer even more pronounced.

Table 5.2: Approximate Uncertainty for different layer types at different depths.

Layer	Shallow	Medium	Deep
Convolutional	1-2	2-3	2-3
Relu	10	20	20
Normalization	800-1000	5-10	4-5
Pooling	4-5	5-6	8-9
Fully Connected	∞	2-3	3-4
Softmax	∞	∞	∞

The second convolutional layer has an Uncertainty that is larger because it is deeper.[2] The Uncertainty indicates that in this case, the power would trail the performance closely by up to 1.7 times. Other layers, particularly the Softmax layer, have high Uncertainty values, indicating that the relationship between power and performance is rather erratic for these layers.

Finally, to ease the burden of future development based on Uncertainty, we offer Table 5.2. Table 5.2 contains an approximate value of Uncertainty for the most common DNN layers. The Uncertainty is based on the two most important factors: the type of the layer and the position of the layer. As discussed earlier, the type of the layer is not the sole contributor to a layer's power/performance behavior. The position of that layer in the neural network is equally important. As the DNN progresses, the amount of information processed by each layer is reduced substantially. On the other hand, the computation required to produce the output for the next layer becomes much more significant. We note that these values of Uncertainty only apply to GPU-based DNNs. Since these types of DNNs are the most widespread version, Table 5.2 should cover most use cases in the real world.

[2]The depth is important because it would reflect on the computational characteristic of the layer. This reflection is mainly due to the input size for that layer. The deeper a layer is, the smaller the input size for that layer is, in most cases.

5.2.2 Progress Tracker

We have explained our intuition behind using Uncertainty to capture the power/performance characteristics of each layer. We now introduce the other component, a progress tracker, that will be integrated into PredJoule to tightly control timing through tracking the execution deficit at each layer boundary.

We define a straightforward progress tracking strategy based on a schedule that is constructed from a randomly selected DVFS configuration. Consider the example shown in Fig. 5.4. Assume there is a DNN instance T consisting of n layers $\{T_1, ..., T_n\}$. Let c_i' denote the execution time of each layer l_i in this schedule. Let ε denote the end-to-end execution deficit in this schedule, i.e., $\varepsilon = F - D$, where F and D are the completion time and deadline of T, respectively. A positive (negative) value of ε implies that T completes after (before) the deadline; the case where $\varepsilon = 0$ implies that T completes exactly at its deadline under this random configuration. At each layer boundary, the system updates the execution deficit and will have to catch up in the case of a positive value. On the other hand, the system might exploit a negative execution deficit for an energy efficiency gain.

Our design tracks the progress of each layer's execution under its runtime configuration C using this random schedule as a reference, which considers the above-defined ε. Our design then compares each layer l_i's execution time under C, denoted as c_i, with the execution time c_i' under the random configuration. That is, from a per-layer perspective, after executing each layer l_i under a certain configuration C, T would have an updated deficit of $\varepsilon = c_i - c_i' + \varepsilon$. A positive (negative) per-layer deficit at each layer boundary implies that after executing layer l_i, the entire schedule is running behind (ahead) compared to the "ideal schedule". From a latency perspective, a perfect scenario is to complete each layer such that the deficit at the corresponding layer boundary is equal to 0, which implies that the entire DNN instance would complete exactly at its deadline.

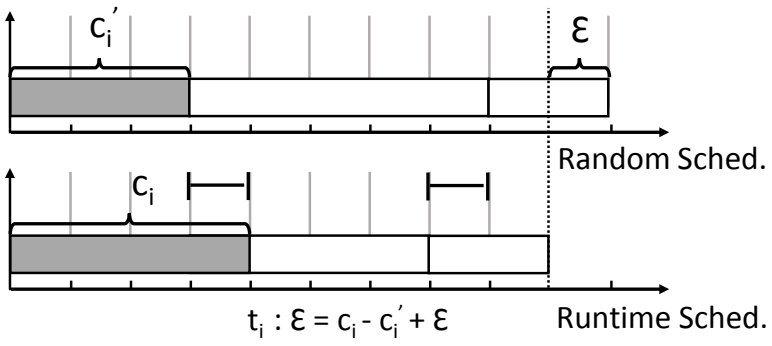


Figure 5.4: Example illustration of the progress tracker.

5.2.3 Integration

In this section, we describe how we integrate our idea of Uncertainty with the progress tracker to achieve timing predictability and energy minimization. Specifically, we consider the execution deficit at each layer boundary and use Uncertainty of the next layer to decide its configuration. There are two cases to consider after executing each layer l_i.

Case 1: $\varepsilon > 0$**.** If the deficit after executing l_i is positive, the system will need to jump into a higher-speed configuration to catch up with the deadline, say from C_i to C_j. This configuration adjustment depends crucially on Uncertainty of the next layer l_{i+1} for energy saving reasons.

As discussed in Sec. 5.2.1, a lower Uncertainty indicates that the layer exhibits a more pronounced trailing effect in which a higher performance results in lower power usage, while a high Uncertainty indicates irregularities that could potentially be inefficient.

Since the configuration space is partially sorted, the jump from C_i to C_j can be achieved by adding a value to i. This would imply that the higher the index value of a configuration is, the faster it will be (we first assume this and correct for any errors once the history becomes reliable). Moreover, a jump to a higher value for configuration index would be directly related to the inverse value of Uncertainty since a small Uncertainty should result in a big jump:

$$j = i + \alpha \frac{1}{U_{l \in L}} i, \tag{5.5}$$

Algorithm 4 Controller component of PredJoule.

Require: configs[] ▷ list of possible DVFS configurations
Require: history[] ▷ speed and power of all conf. compared to c_r
1: **function** CONTROLLER(curr_c, expected_deadline)
2: Update(ε)
3: req._speedup ← expected_deadline / (expected_deadline+ε)
4: c ← $\min_{powerup}$ (LookUP(history, $req.\_speedup$))
5: **if** $\varepsilon < 0$ **then**
6: **if** c **and** M> $threshold_M$ **then** ▷ M: Maturity
7: **return** c
8: **else**
9: c ← configurations[$\alpha.\frac{1}{U_l}.indexOf(curr\_c)$]
10: **else if** $\varepsilon \geq 0$ **then**
11: **if** c.powerup < curr_c.powerup **and** M> $threshold_M$ **then**
12: **return** c
13: **else**
14: c ← configurations[$\beta.U_l.indexOf(curr\_c)$]
15: **return** c

in which both c_i and c_j are part of the configuration space C. The variable α acts as a user-defined gauge that can control the aggressiveness of Eq. 5.5.

Case 2: $\varepsilon \leq 0$**.** In the case that the deficit is less than or equal to zero, PredJoule will try to exploit the resulting negative ε for gaining a more optimal energy configuration. In other words, if the Uncertainty of l_{i+1} is high, it is possible to gain energy optimization by running slower. For example, an infinite Uncertainty implies that the slowest configuration should be used to be conservative. We reuse Eq. 5.5 with a reversed coefficient:

$$j = i - \beta U_{l \in L} i. \tag{5.6}$$

β is the corresponding coefficient for a negative execution deficit.

A Learning-based Approach. An assumption made in Eqs. 5.5 and 5.6 is that the configuration space C is perfectly sorted according to the speedup gained. However, in real-world, there could be oddities present in a partially sorted set. Moreover, Uncertainty tries to encapsulate how each layer would react to a configuration change into just one number, which will obviously result in

a loss of detail that can potentially hamper the process of improving the energy usage. Finally, it is impossible to guarantee any latency requirement because the controller cannot predict the exact degree of speedup a new configuration would bring. If there were an exact speedup associated with the new configuration, the controller would have been able to intricately choose a configuration that would exactly match the current system need.

Contemporary literature concerning this matter always resorts to some sort of offline benchmarking in order to mitigate this problem (Imes et al., 2015; Zhang and Hoffmann, 2016; Mishra et al., 2015). While offline benchmarking can be effective in isolation, it cannot work for a runtime environment. The biggest reason is that an offline solution will not be able to adapt to any interference due to either workload or system environment changes. This is a major drawback even without considering the overhead of offline benchmarking for hundreds of layers run under thousands of configurations.

Rather, we would like to exploit a special characteristic of neural networks: All neural networks are constructed from a limited set of layers and keep a consistent configuration throughout multiple executions. Since the parameters, operations and data width and types are consistent between each run, variations in input cannot affect the execution behavior of a DNN in a measurable way. This motivates us to design a learning-based approach that can learn and adapt more granularity over time. Such an approach is especially desirable in applications such as autonomous driving in which a learning procedure is always part of the development. For example, Tesla uses a "fleet learning" procedure (Tesla, 2017) to make vehicles learn driving from shared driving data.

For the learning phase, we introduce a history recorder to PredJoule. Each time the controller selects a new configuration according to Eq. 5.5 and Eq. 5.6, it will record the resultant speed and energy change for that specific layer. To make the history data structure and the lookup procedure simple, all the changes are recorded by dividing the newly measured power and speed of the system by a single base configuration: the random configuration first defined in Sec. 5.2.2. Note that history is layer-based.

However, the accuracy of the history entirely depends on the number of new configurations that have been explored by using Eq. 5.6 and 5.5. If the system has not been running for long or has settled on an optimal configuration after visiting only a few configurations, the history can be quite unreliable. The only solution in this scenario would be to continue to explore new configurations.

We represent this quality by a variable called system maturity denoted by M. For a DVFS configuration space C and a history lookup table H, the maturity of the system is defined as:

$$M = \frac{|H|}{|C|}, \tag{5.7}$$

in which M is always between 0 and 1. The $|H|$ and $|C|$ depict the number of entries in H and C, respectively. The problem now becomes that of setting a threshold. The closer the M is to 1, the more likely it is for PredJoule to choose a configuration already in the history. We set an extremely high value of the threshold (0.9999) because in our experiments, the system does not miss the deadline after a few iterations. We note that M is layer based because history is layer based.

The overall steps for PredJoule controller are depicted in Algorithm 4. The LookUP procedure is for searching history. Imagine a neural network extracted from the example of Table 5.1. For the sake of simplicity, we only assume three layers exist: Conv1, Relu1, and Softmax. First, PredJoule will choose a random DVFS configuration (e.g., c_{400}). It will then run a warmup run using that configuration and record the execution time of each layer. After the warmup run is finished, the controller will calculate ε by deducting the deadline from the total execution time. For example, with a deadline of 30ms and total execution time of 35ms, ε will initially be 5. The controller will also set the deadlines of each layer to the recorded execution times. This deadline will be used to update the ε and calculate the speedup for subsequent runs of the neural network.

With a positive deficit, the system will need to catch up. At the beginning of next execution, the controller (at line 5) will recognize the positive ε. Moreover, the maturity is just 1(the random configuration) divided by the configuration space size (1411 in our case). The check thus will fail

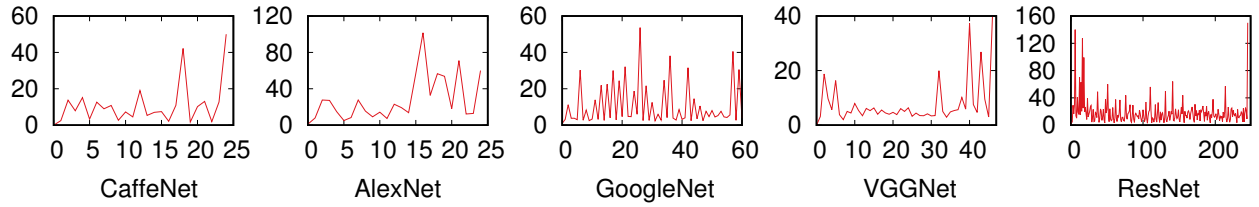


Figure 5.5: Uncertainty for various neural networks.

at line 6. For this layer's boundary, the system will use Eq. 5.5 in line 9 instead, with an α of 1. The system will jump to $c_{400+1.1/U_l.400}$, which is c_{900}. At the boundary of layer 1, the controller will record the speedup of c_{900} (based on c_{400}). If ε is still positive, the same procedure is applied. But let us assume that the system has caught up with the execution deficit. Then, at the boundary of layer 2, with a maturity of $1/1411$ (for layer 2), the system will use Eq. 5.6 (at line 14) instead. This equation backtracks any jump in order to examine previous configurations that have been missed. For layer 2 and a β of 0.001, the new configuration will be $c_{400-0.001.10.400} = c_{360}$ (β is always much smaller than α because catching deadline is much more urgent than exploration). This procedure continues until the system reaches enough maturity so that it can solely rely on history. While the deadline can be caught before then (as it almost always does in our experiments), PredJoule can always meet the required predictability once the system is mature.

5.3 Evaluation

5.3.1 System Setup

We fully implement PredJoule and perform an extensive set of experiments on a cutting-edge autonomous driving embedded platform, which is an NVIDIA Jetson TX2. This platform has a 6-core big.LITTLE CPU with 2 large Denver cores and 4 ARM A57 cores. The GPU is a Pascal-based 256 core processor that shares 8GB of RAM with the main CPU. The GPU and the CPU are embedded in an NVIDIA Parker system-on-a-chip

We intend to evaluate PredJoule's ability to provide timing predictability and energy savings. We compare PredJoule to the following approaches:

• **Poet** (Imes et al., 2015): Poet is an open-source library that uses a control theory approach to calculate deadline errors and decides on a speed-up based on that error. For energy efficiency, the authors of Poet have added an optimizer that loops with an order of $O(N^2)$ on all configurations to find two configurations: one configuration that runs slower but is energy efficient and one configuration that is fast and inefficient. This optimizing algorithm is done in order to find the best balance between energy and latency. Another approach, MEANTIME (Farrell and Hoffmann, 2016) is also concerned with energy efficiency. However, it is based on Poet with the addition of approximation. With no approximation, MEANTIME may not be effective.

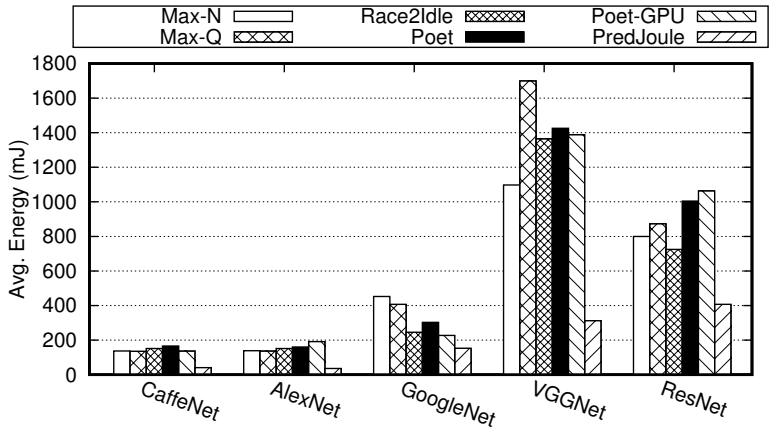
• **Poet-GPU**: Poet does not consider platforms containing GPUs. We thus have implemented a version of poet that can control GPU's DVFS configuration in addition to CPU.

• **Race2Idle**: Race2Idle (Kim et al., 2015) is the infamous simplistic approach to meet hard deadlines. The premise is simple: run everything at max frequency with all cores activated. While this method is easy to dismiss because it might be assumed to have bad energy efficiency, it beats Poet in some scenarios according to our observations.

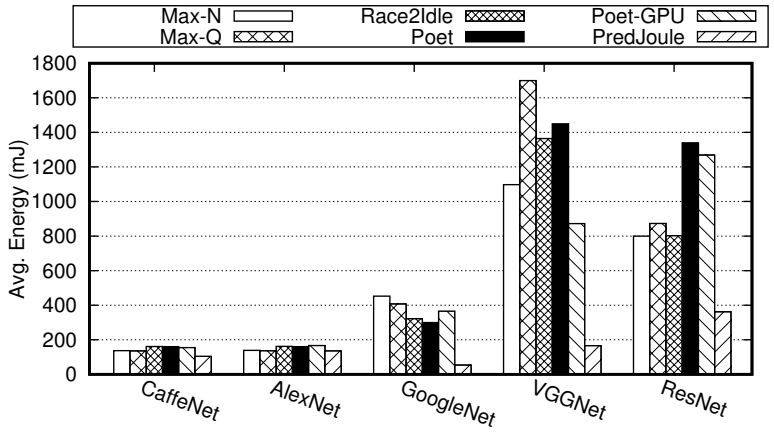
• **Max-N**: NVIDIA has implemented several hardware DVFS modes that are preloaded on Jetson TX2 and are accessible using the nvpmodel command. Max-N is the high performance mode that allows for the maximum frequency of CPU and GPU. However, the frequencies are adjusted dynamically at runtime.

• **Max-Q**: Max-Q is the most energy efficient mode that disables big cores entirely. Moreover, it caps the maximum frequency on both CPU and GPU for energy saving purposes.

We test each of the six approaches using five different DNN models: GoogleNet, CaffeNet, AlexNet, VGGNet, and ResNet. We first show the general usability of PredJoule by comparing it against other methods on all five DNN models, under two deadline configurations of tight and relatively loose deadlines. Next, we use the largest and the most advanced among these neural networks, which is ResNet, to demonstrate detailed latency and energy results under two scenarios: running DNN workloads with and without interference. We end by measuring PredJoule's overhead.



(a) Tight deadline



(b) Loose deadline

Figure 5.6: The average energy consumption of all five neural networks with tight / loose deadlines.

5.3.2 Generality

First and foremost, we test the deadline miss ratio and energy usage of all six methods for running the five DNN models with two deadline settings: tight and loose deadlines. According to the specific computational demand, we set a tight (loose) deadline of 20ms / 20ms / 50ms / 50ms / 100ms (25ms / 25ms / 100ms / 100ms / 150ms) for CaffeNet, AlexNet, GoogleNet, VGGNet, and ResNet, respectively.

We test five different neural networks because each has a unique combination of layers with different sizes and depths. Fig. 5.5 shows the differences of the five neural networks in terms of the value of Uncertainty for different layers. As is evident in the figure, CaffeNet and AlexNet are

Table 5.3: Method deadline misses for various DNNs.

	CaffeNet	AlexNet	GoogleNet	VGGNet	ResNet
Max-N	0%	0%	100%	100%	100%
Max-Q	100%	100%	100%	100%	100%
Race2Idle	0%	0%	0%	100%	0%
Poet	1%	0%	4%	99%	16%
Poet-GPU	51%	9%	6%	99%	51%
PredJoule	0%	0%	10%	3%	0%
After 50 iter.	0%	0%	0%	0%	0%

relatively small networks. The most fluctuation for these neural networks is happening towards the end since that is where fully connected and softmax layers are located with small input sizes and large computations. While the same is true for VGGNet and ResNet, they are much more complicated, especially ResNet with over 200 layers. However, these extra layers are added in the name of improved accuracy. Finally, GoogleNet has a concentration of Relu layers in the middle that show up as high points of Uncertainty.

As is depicted in Fig. 5.6(a), PredJoule outperforms all other methods by a significant margin in most cases. It is observed that Race2Idle can perform as good as Max-Q or better. This is due to the fact that if a DNN is considered as a layer-oblivious blackbox, then it will have a low Uncertainty overall and will thus benefit from running faster. However, running faster will have a potentially adverse effect on layers with a high Uncertainty. Thus, our method can get much more efficient than Race2Idle by scaling back the DVFS for layers with high Uncertainty values. Finally, Poet does not perform well because it neither controls GPU nor being layer-aware. Our own port of Poet to GPU performs better than Poet because it controls GPU frequency for energy optimization. However, the effect of a layer-based design exploring layer characteristics is clearly depicted when comparing Poet-GPU to PredJoule.

The second scenario depicted in Fig. 5.6(b) shows the same experiment but under a loose deadline setting. Having a loose deadline can show the efficacy of exploiting the system idleness

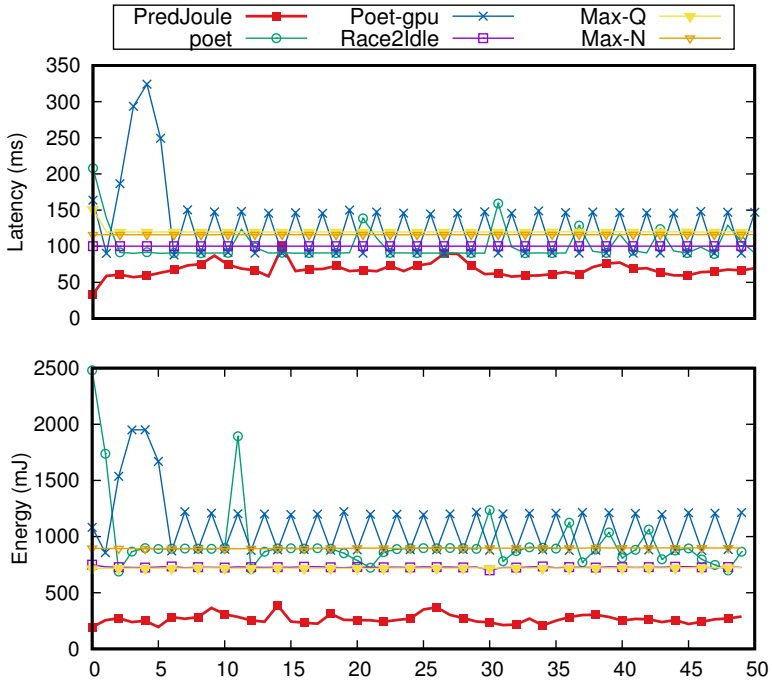


Figure 5.7: Energy and latency of PredJoule compared to others for ResNet-100 over 50 iterations.

under each method. For example, Race2Idle will suffer more when the system is idle for longer since its configuration does not change between scenarios. Other methods may suffer less because they switch to different settings accordingly. For two DNNs (VGGNet and ResNet), Max-N will be better than Max-Q in terms of energy efficiency. This scenario happens because the speed gain from running VGGNet and ResNet under Max-N will outweigh the added energy usage of it. Moreover, reactive methods such as Poet/Poet-GPU should be able to utilize looser deadlines to minimize energy since they make tasks run longer and may achieve better energy efficiency. While PredJoule benefits from a looser deadline compared to the tight case, Poet clearly cannot. This is due to the fact that Poet does not consider idleness as an adverse effect on energy and will not try to remedy it. Our method, on the other hand, will backtrack on high Uncertainty layers to find more efficient configurations. This effect is the most pronounced in VGGNet, which has a complicated structure.

Finally, we compare the deadline miss ratios of PredJoule against the other five methods in Table 5.3. We run each task for 1000 iterations with the tight deadline and record a missed deadline.

PredJoule can outperform others in almost all cases. For CaffeNet, AlexNet, and ResNet our method never misses a deadline. This is because PredJoule is able to recover the execution deficit inside the first iteration, rather than waiting until after the first iteration. However, our method misses 10% of deadlines for GoogleNet and performs worse than Race2Idle, Poet, and Poet-GPU in this specific scenario. This adverse effect is because GoogleNet contains many Relu layers that have a high Uncertainty. Thus, our method will take longer to adapt and meet the deadline, missing deadlines only at the beginning. We showcase the execution of our method compared to the others in a subsequent section. Finally, for the complicated VGGNet, our method only misses 3% of deadlines, which is much better than others. In fact, for VGGNet and our tightly set deadline, other methods will almost entirely miss their deadlines. This is a special case in which our layer-based design would shine the most, because VGGNet has many layers with low Uncertainty, even more so than CaffeNet and AlexNet. Nonetheless, our method still misses deadlines at the beginning and is not able to recover inside the first iteration. We believe our method can improve on timing predictability compared to competing approaches, as is evident by no deadline misses after 50 iterations.

5.3.3 Detailed Latency/Energy Performance

We now showcase a detailed set of latency and energy results for all the six methods over time for running ResNet with a deadline of 100ms for 50 iterations. ResNet is the largest and the most advanced among our tested neural networks. As seen in Fig. 5.7, other than Race2Idle, our method is the only one that can meet the latency requirements of ResNet. Regarding energy efficiency, our method also outperforms the other methods by a considerable margin. This set of results demonstrate that our design and implementation of PredJoule can achieve two (often) conflicting goals of timing predictability and energy efficiency. The consistency for Max-Q and Max-N are also apparent in this figure. However, they cannot match the latency of our method. Moreover, PredJoule can be as much as 66% and 70% more efficient compared to Max-N and Max-Q.

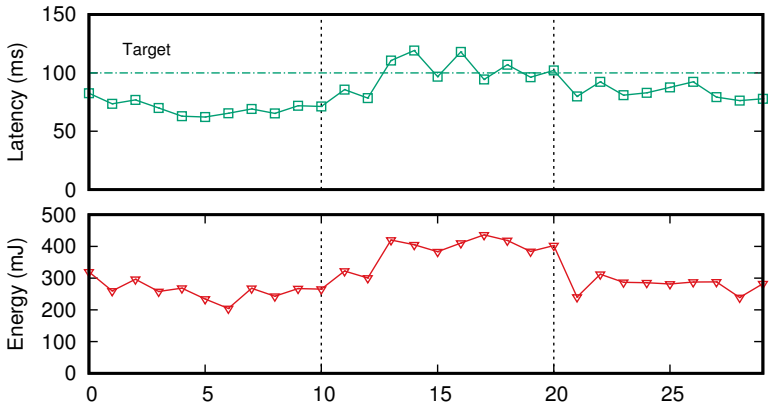


Figure 5.8: Energy and latency of PredJoule when interference is present.

5.3.4 Adaptability With Interference

While the energy and latency can be optimized under PredJoule when the neural network is run as a standalone application, this scenario may not always happen in practice. In many cases, the neural network is accompanied by other tasks that run concurrently. For example, in autonomous driving, it is possible that an object tracking task is running in parallel to a route planning task. In this experiment set, we test our method's adaptivity with the presence of interference.

Fig. 5.8 shows the detailed energy and latency results for running ResNet with a deadline of 100ms under our method over 30 iterations. At iteration 10, we manually add interfering workloads. The interference will finish execution at iteration 20, as depicted by the cross lines. As seen in the figure, when the interference is introduced, the added energy and latency will initially hurt the execution of ResNet. Nonetheless, PredJoule is able to recover rather quickly and meet the deadline within a few iterations. Similarly, the energy usage will only increase slightly when the adaptation happens. PredJoule is able to respond to interference because it actively updates the execution deficit ε at each layer boundary. ε shall include any interference because it is always calculated in a real-time fashion in our implementation. Also, PredJoule is able to recover inside 10 iterations, making the execution more responsive.

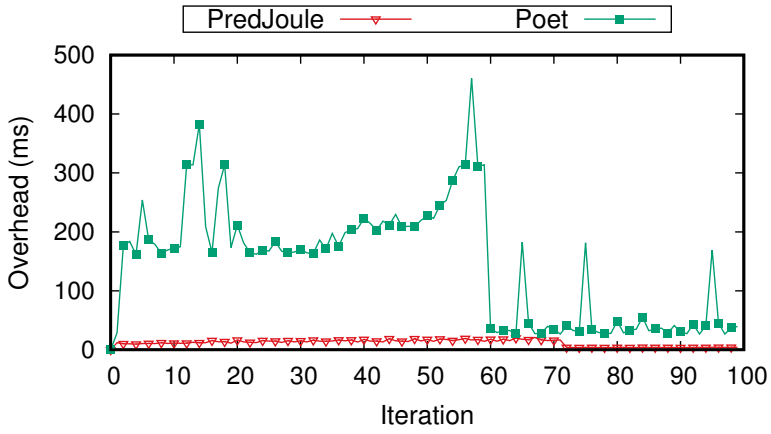


Figure 5.9: Overhead of PredJoule vs. Poet for ResNet.

5.3.5 Overhead

While it might be tempting to ignore the overhead for energy saving solutions, we believe it is crucial for two reasons. First, our method aims at meeting latency requirements. Thus, a solution with a high overhead may defeat this purpose. Second, most energy-related literature might assume that frequency change and DVFS, in general, are free. Our findings show that DVFS may not have a negligible overhead, especially for Linux since it requires multiple file accesses. In this set of experiments, we evaluate the runtime overhead incurred under PredJoule to assess whether our design and implementation is efficient in practice.

We compare our method against Poet for running ResNet with a deadline of 100ms, in terms of runtime overhead due to DVFS configuration adjustment (note that Poet is shown to be a rather efficient runtime DVFS solution (Imes et al., 2015)). As is depicted in Fig. 5.9, PredJoule is highly efficient even compared to Poet. This efficiency comes from two factors: First, PredJoule implements an efficient implementation of DVFS for Linux that relies solely on file handles and does not need to perform costly file accesses. Second, Poet iterates through all configurations before finding an optimal configuration. As is evident in Fig. 5.9, at around iteration 60, Poet has found an optimal configuration. However, the fluctuation in overhead still remains as Poet does a preliminary calculation before moving on to the next iteration. On the other hand, PredJoule

mostly uses Eq. 5.5 and 5.6 which are done in constant time. For occasions that the history might be needed, we implement an efficient reverse history that gives the desired speed up in $O(1)$ time. Once our method has found an efficient configuration, the overhead becomes minimal since there is no need for DVFS operations any longer.

5.4 Summary

This chapter presents PredJoule, a timing-predictable energy optimization framework for running DNN workloads in a GPU-enabled automotive system. PredJoule presents a layer-based approach that controls the timing and optimizes energy efficiency through exploiting each layer's performance/energy characteristics. Our experimental results demonstrate that PredJoule is able to achieve timing predictability with no deadline misses and much higher energy efficiency than prominent methods.

CHAPTER 6

TASK MAPPING IN HETEROGENEOUS SYSTEMS FOR FAST COMPLETION[1]

6.1 System Modeling and MIP Formulation

In this section we give out a list of notations and definitions to help us better formalize the proposed problem, and then we propose a MIP (Mixed Integer Programming) Formulation to get optimal mapping in theory.

6.1.1 System Model

Let us consider the problem of mapping n independent applications $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$ onto m processors $M = \{M_1, M_2, ..., M_m\}$. Each processor is either a CPU or a GPU.

Each application τ_i is composed by serial instructions and kernels, where kernels represent computation operations. Kernels of each application are chained together according to the computation logic. They may have dependencies since data flows from one kernel to another. That is, τ_i is modeled as a kernel graph that contains z_i connected kernels $\{\tau_i^1, \tau_i^2, ..., \tau_i^{z_i}\}$. Let N denote the total number of kernels of applications in Γ. Each kernel τ_i^j has an execution time of $C_{i,k}^j$ if executed on processor p_k. The execution time ranges from milliseconds to hours, depending on the specific application. Similar to prior work (Luk et al., 2009), we use the sampling functionality of StarPU (Augonnet, Thibault, Namyst, and Wacrenier, Augonnet et al.) to obtain the estimated execution time of a kernel.

Between any two connected kernels is an edge, which implies that precedence constraints exist between these two kernels. If kernel τ_i^j has an outgoing edge e_i^{jk} to kernel τ_i^k, then τ_i^k cannot start execution until it receives the data produced by τ_i^j. Let $\mathcal{P}(\tau_i^j)$ denote the set of predecessor kernels of τ_i^j, i.e., kernels that have outgoing edges from τ_i^j. Similarly, let $\mathcal{S}(\tau_i^j)$ denote the set

[1]

Table 6.1: Notation Summary.

N	number of total tasks
n	number of applications
m	number of processors
τ_i	i^{th} application
τ_i^z	z^{th} kernel/task of application τ_i
M_i	i^{th} processor (either a CPU or a GPU)
$C_{i,k}^j$	execution time of τ_i^j on processor M_k
$\mathcal{S}(\tau_i^j)$	set of successor kernels/tasks of τ_i^j
$\mathcal{P}(\tau_i^j)$	set of predecessor kernels/tasks of τ_i^j
e_i^{jk}	edge from τ_i^j to τ_i^k
$T_{q\to w}(e_i^{jk})$	time taken to send data from τ_i^j to τ_i^k

of successor kernels of τ_i^j, i.e., kernels that have incoming edges to τ_i^j. Let $T_{q\to w}(e_i^{jk})$ denote the time for τ_i^j executed on processor p_q to send its produced data to its successor τ_i^k (connected by edge e_i^{jk}) executed on processor p_w. A summary of important notations is given in Table 6.1. We use the term *task* to represent a kernel combining with its needed data. For readability, in the rest of this chapter, we will use *task* and *kernel* interchangeably.

Definition 1. *We define the depth of a kernel to be the number of kernels on the longest path between this kernel and a kernel of the corresponding kernel graph that has no predecessors. Kernels with no predecessors have a depth of 1. Let $D(\tau_i^z)$ denote depth of kernel τ_i^z in the kernel graph of τ_i.*

Preemptive vs. non-preemptive execution On GPUs, executions are often non-preemptive (Elliott and Anderson, 2011). That is, once a kernel starts execution on a GPU, it cannot be preempted by other kernels until its completion. On CPUs, executions can be preemptive. However, preemptions may incur significant amount of overheads at runtime such as context switch overhead and migration overhead (Basaran and Kang, 2012). To ensure the efficiency, as well as to simplify the formalism and algorithms, we thus assume in this chapter non-preemptive executions on CPU as well.

6.1.2 An MIP Formulation

We formalize the problem of minimizing the makespan of a given set of applications executed on m CPU and GPU devices by specifying an integer program with a polynomial number of variables as follows. We first define a new set of variables that are used in the integer program.

Definition 2. *Define*

$$p_w^x(\tau_i^j) = \begin{cases} 1 & \text{if } \tau_i^j \text{ is the } x\text{-th kernel executed by } M_w \\ 0 & \text{otherwise,} \end{cases}$$

for all $1 \leq w \leq m$, $1 \leq x \leq n$, $1 \leq i \leq n$, $1 \leq j \leq z_i$. *For every kernel* τ_i^j, *let* $s_i^j \geq 0$ *denote the starting time of its execution.*

The makespan can be denoted by C_{max}. Then, this problem may be formulated as:

minimize C_{max}

subject to:

$$\sum_{q=1}^{m}\sum_{g=1}^{N} p_q^g(\tau_i^j) = 1, \qquad \forall \tau_i^j \in \Gamma \tag{6.1}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{z_i} p_q^1(\tau_i^j) \leq 1, \forall q = 1, ..., m \tag{6.2}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{z_i} p_q^g(\tau_i^j) \leq \sum_{i=1}^{n}\sum_{j=1}^{z_i} p_q^{g-1}(\tau_i^j), \forall q = 1, ..., m, \forall g = 2, ..., n \tag{6.3}$$

$$\begin{aligned} s_i^j \;\geq\; & s_i^k + \sum_{q=1}^{m}\sum_{g=1}^{N} C_{i,q}^k \cdot p_q^g(\tau_i^k) \\ + \; & \sum_{q=1}^{m}\sum_{g=1}^{N}\sum_{v=1}^{m}\sum_{h=1}^{N} p_{i,q}^g(\tau_i^j) \cdot p_{i,v}^h(\tau_i^k) \cdot T_{q\to v}(e_i^{jk}), \\ & \forall \tau_i^j \in \Gamma, \forall \tau_i^k \in \mathcal{P}(\tau_i^j) \end{aligned} \tag{6.4}$$

$$s_i^j \geq s_l^k + C_{l,q}^k - \alpha \cdot \left(2 - \left(p_q^g(\tau_l^k) + \sum_{r=g+1}^{N} p_q^r(\tau_i^j)\right)\right) \quad (6.5)$$
$$\forall \tau_i^j \in \Gamma, \forall \tau_l^k \in \Gamma,$$
$$\forall q = 1, 2, ..., m, \forall g = 1, 2, ..., N-1$$

$$C_{max} \geq s_i^j + \sum_{q=1}^{m} \sum_{g=1}^{N} C_{i,q}^j \cdot p_q^g(\tau_i^j), \forall \tau_i^j \in \Gamma \quad (6.6)$$

$$s_i^j \geq 0, \forall \tau_i^j \in \Gamma \quad (6.7)$$

$$p_q^g(\tau_i^j) \in 0, 1, \forall q = 1, ..., m, \forall g = 1, ..., N, \forall \tau_i^j \in \Gamma \quad (6.8)$$

where $\alpha >> 0$ is a sufficiently large penalty coefficient.

Integer program description. Eq. (6.1) ensures that each task is assigned to exactly one processor. Eq. (6.2) ensures that at most one task will be the first one to be executed by any given processor. If a task is the g^{th} (where $g \geq 2$) assigned to processor M_q, then there must be another assigned as the $(g-1)^{th}$ task of this same processor, as ensured by Eq. (6.3). Moreover, Eq. (6.4) ensures that precedence constraints are respected. That is, no task τ_i^j may start execution unless all its predecessors have already completed their execution and τ_i^j has already received the data produced by its predecessors.

Eq. (6.5) defines the sequence of starting times of the set of tasks assigned to the same processor. It expresses the fact that task τ_i^j must start at least $C_{l,q}^k$ time units after the beginning of task τ_l^k, whenever it is executed after task τ_l^k on the same processor M_q, i.e., $p_q^g(\tau_l^k) = \sum_{r=g+1}^{N} p_q^r(\tau_i^j) = 1$ for some $g = 1, 2, ..., N-1$. Eq. (6.6) defines the constraint on the makespan (i.e., the maximum completion time among all kernels).

By solving the above formulation, we obtain an optimal solution that minimizes the makespan. Unfortunately, solving this integer program (although it has a polynomial number of 0-1 variables) is quite expensive in practice. In the next sections, we report several key observations motivated by measurements-based case studies (Sec. 6.2), which further motivate our design on several efficient online mapping algorithms that can be applied in practice (Sec. 6.3).

6.2 Case Studies: What to Consider for Making Mapping Decisions

In this section, we present several measurements-based case studies that motivate the design of our mapping algorithms. We measured the completion time of executing a vector add application τ_1 and a matrix multiplication application τ_2 on a heterogeneous system configured with one Intel Core i7 CPU and NVIDIA GeForce GTX660 GPU. τ_1 can be expressed as $(v_1 + v_2) * \pi$, where v_1 and v_2 are vectors and π is a constant. τ_2 can be expressed as $(a * b) + (c * d)$, where a, b, c, d are four input matrices. These applications are commonly seen in scientific computing. The corresponding kernel graphs are illustrated in Fig. 6.1. Specifically, τ_1 contains two kernels τ_1^1 τ_1^2, where τ_1^1 is a vector add kernel and τ_1^2 is a vector scale kernel. τ_2 contains three kernels, where τ_2^1 and τ_2^2 are two matrix multiplication kernels, and τ_2^3 is a matrix add kernel. For the generated input data, v_1 and v_2 have a size of 50000 elements each. a, b, c, and d are four matrices with a size of $1024 * 1$, $1 * 1024$, $1024 * 1024$, and $1024 * 1024$, respectively. Through profiling, the execution time of each kernel is listed in Table 6.2. We have conducted various experiments based upon this system setup and recorded the corresponding mapping sequences and completion times under different strategies. Among the obtained results, we have identified several factors that may significantly affect the mapping performance.

Observation #1: kernel-level mapping or application-level mapping? In this case study, our observation is that for applications that contain multiple dependent kernels, treating kernels as the mapping entity yields better performance than mapping each entire application to a processing unit. Fig. 6.2(a) shows the schedule of performing application-level mapping. The dash lines in

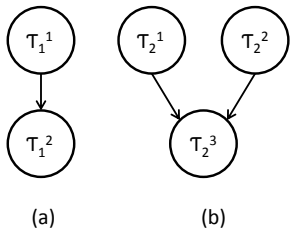


Figure 6.1: Kernel dependency graph

Table 6.2: Execution time of kernels

	CPU	GPU
τ_1^1	$5.68 \times 10^2 \mu$s	$4.22 \times 10^2 \mu$s
τ_1^2	$1.52 \times 10^3 \mu$s	$2.42 \times 10^2 \mu$s
τ_2^1	$4.41 \times 10^4 \mu$s	$5.6 \times 10^3 \mu$s
τ_2^2	$8.74 \times 10^2 \mu$s	$8.44 \times 10^2 \mu$s
τ_2^3	$4.40 \times 10^2 \mu$s	$4.20 \times 10^2 \mu$s

this figure represent the final completion time. The (tiny) space among kernel execution blocks represents the delay due to necessary data transfer. Fig. 6.2(b) shows the schedule of performing kernel-level mapping, in which we can see that the completion time is shortened. The main performance acceleration comes from the parallel executions of multiple kernels on two processing units. Intuitively, for systems that support multiple applications, kernel-level mapping is a better choice because it can better utilize the hardware resources.

Observation #2: heterogeneity matters. Fig. 6.2(c) shows the schedule of a kernel mapping policy with a different kernel ordering scheme than the mapping policy shown in Fig. 6.2(b). The applied mapping policy considered in this case prioritizes kernels by considering the heterogeneity. Intuitively, a kernel that has a faster execution time on a specific type of processor (either CPU or GPU) should preferably be assigned to that type of processor. As shown in Table 6.2, kernels τ_1^2, and τ_2^1 have much shorter execution times on GPU compared to CPU. Thus, by prioritizing such kernels over other kernels (such as τ_1^1 and τ_2^2), they have higher possibilities to be assigned to their favorite processors, as observed in Fig. 6.2(c). This case study highlights the fact that for

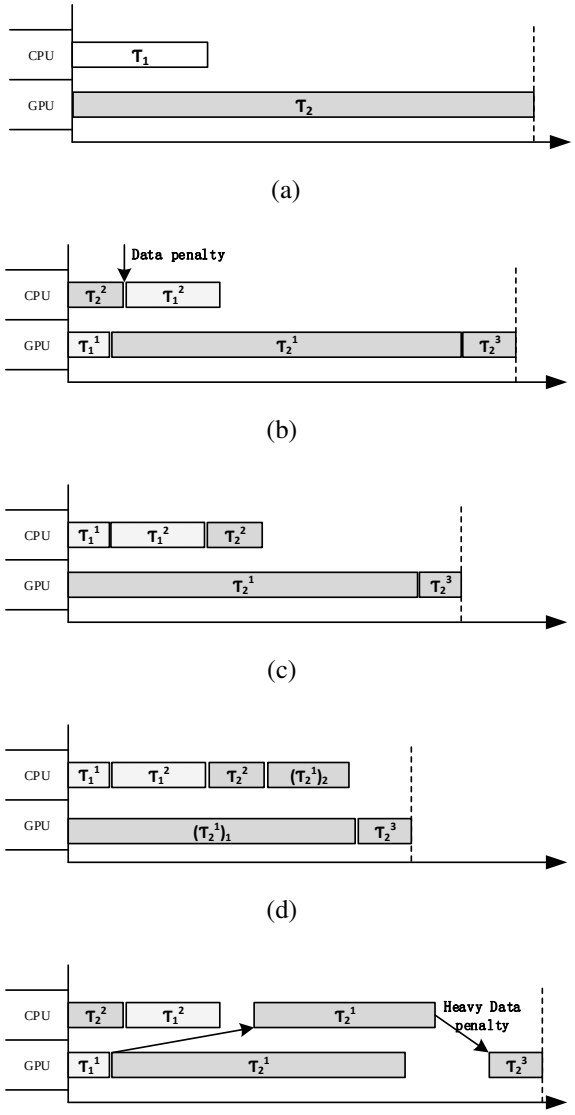


Figure 6.2: (a) Application level mapping and (b) Kernel level mapping (c) Different map order (d) Data Partition (e) Bad data partition

CPU/GPU systems, the heterogeneity reflected by hardware and application characteristics must be considered in the mapping algorithm.

Observation #3: data partitioning—is it always beneficial? As seen in Table 6.2, τ_2^1 is the most computation-intensive kernel. Fig. 6.2(c) shows that τ_2^3 cannot start execution because τ_2^1 completes late, which causes resource under-utilization and longer completion times. By partitioning

the input matrix of τ_2^1 into two slices, we are able to reduce its execution time by running the kernel with partial data on both CPU and GPU in parallel. Let $(\tau_2^1)_1$ and $(\tau_2^1)_2$ denote the resulting two kernels each with half data. The resulting schedule with reduced completion time is shown in Fig. 6.2(d). However, data partitioning is not free. It incurs additional data transfer overhead because data need to be sent to both $(\tau_2^1)_1$ and $(\tau_2^1)_2$, and the corresponding results need to be merged and then sent to τ_2^3. Since the data size is not very large in this case, the performance gain due to data partitioning overwhelmed the penalty due to additional data transfer. Nevertheless, when we increase the input matrix size for τ_2^1 to 16384 * 16384, the negative impact due to additional data transfer under partitioning becomes obvious, as illustrated in Fig. 6.2(e). Our observation herein is that data partitioning may be beneficial only when the input data size is reasonably small.

It is clear from these case studies that the completion time performance heavily depends on the mapping algorithm, which needs to consider a number of influential factors including the kernel structure, heterogeneity, kernel prioritization, and data partitioning.

6.3 Practical Mapping Algorithms

In this section, we present three practical online algorithms for mapping tasks in a heterogeneous platform consisting of multiple CPUs and GPUs. Our algorithmic design is motivated by the observations as discussed in Sec. 6.2. Specifically, the proposed mapping algorithms consider heterogeneity, kernel graph structure, and data partitioning. The first algorithm (we call it the baseline algorithm) mainly factors heterogeneity into making mapping decisions (besides considering traditional factors such as data locality and earliest completion time). The second algorithm considers kernel structure when prioritizing tasks. The third algorithm extends the baseline algorithm by taking advantages of data partitioning. As seen in Sec. 6.4, these three algorithms yield different performance under different experimental scenarios, depending on specific application characteristics.

6.3.1 Baseline Algorithm: Heterogeneity Ratio-based Mapping

As discussed in Sec. 6.2, without considering heterogeneous workload characteristics on CPUs and GPUs, the mapping algorithm is unlikely to efficiently utilize the heterogeneous resources. Our proposed baseline algorithm takes heterogeneity into consideration when making mapping decisions. Before describing the algorithm, we first give several definitions.

Definition 3. *The **favorite ratio** $F_{i,k}^j$ of a task τ_i^j executed on processor M_k is defined to be*

$$F_{i,k}^j = \frac{max_{h=1}^m(C_{i,h}^j)}{C_{i,k}^j} \tag{6.9}$$

For any task τ_i^j, a larger $F_{i,k}^j$ value implies τ_i^j is more suitable to be executed on M_k. That is, τ_i^j may have a shorter execution time if executed on M_k compared to other processors.

Definition 4. *The **heterogeneity ratio** of a task τ_i^j is defined to be*

$$H_i^j = max_{k=1}^m(F_{i,k}^j) \tag{6.10}$$

For any task τ_i^j, a large heterogeneity ratio implies that it may be more beneficial to execute τ_i^j on one of its favorite processors M_k where $F_{i,k}^j$ is large.

Example: Considering the example system described in Sec. 6.2, the *favorite ratio* of τ_1^1 if executed on processor 1 (CPU) is $F_{1,1}^1 = max(C_{1,1}^1, C_{1,2}^1)/C_{1,1}^1 = 5.68/5.68 = 1$, and the favorite ratio of τ_1^1 if executed on processor 2 (GPU) is $F_{1,2}^1 = max(C_{1,1}^1, C_{1,2}^1)/C_{1,2}^1 = 5.68/4.22 = 1.35$. The heterogeneity ratio of τ_1^1 can be calculated by $H_1^1 = max(F_{1,1}^1, F_{1,2}^1) = F_{1,2}^1 = 1.35$.

Definition 5. *Let $MDAC(\tau_i^j, M_q)$ denote the **Max Data Transfer Time** of τ_i^j if τ_i^j is assigned on M_q, which is defined as the maximum time for transferring data from any of τ_i^j's predecessor tasks to τ_i^j. Specifically, $MDTT(\tau_i^j, M_q)$ is given by*

$$MDAC(\tau_i^j, M_q) = \max_{\tau_i^k \in \mathcal{P}(\tau_i^j)} T_{g \to q}(e_i^{kj}) \tag{6.11}$$

where τ_i^k is executed on M_g.

Definition 6. *Let $EFT(\tau_i^j, M_q)$ denote the **Earliest Finish Time** of τ_i^j if τ_i^j is assigned on M_q. It is defined as:*

$$EFT(\tau_i^j, M_q) = T_{Avail}(M_q) + MDAC(\tau_i^j, M_q) + C_{i,q}^j \tag{6.12}$$

where $T_{Avail}(M_q)$ is the earliest time at which processor M_q is available,

Our proposed baseline algorithm prioritizes tasks based on their heterogeneity ratio. The intuition is to give tasks with larger heterogeneous ratios higher possibilities to be assigned on their favorite processing units. Computing each task's heterogeneity ratio at runtime may incur a considerable amount of overheads. To avoid such overheads, in our implementation, we maintain a lookup table for each task, which records its historical sampling information. Consider the matrix multiplication kernel as an example. Each entry in the lookup table contains data size, average execution time, processing unit to which it is assigned, heterogeneity ratio, hash value, etc. Thus, at runtime, we only need to check the lookup table to figure out the needed information (e.g., heterogeneity ratio). After prioritizing tasks, the algorithm selects the best processing unit for executing each task in turn based on the earliest finish time. The psuedo-code of the algorithm is given in Algorithm 5.

Pseudo-code description. The *PushTask()* function on Line 1 is in charge of pushing incoming tasks into the ready queue of the scheduler. It first obtains the heterogeneity ratios from the lookup table for each incoming task (Line 2), then inserts the tasks into the ready queue by largest-heterogeneity-ratio-first (Lines 3-8). On Line 9, function *GetAllDeviceLen()* gets the total number of assigned tasks in all device queues. If the number is less than a predefined threshold thr (Line 10), then the scheduler executes the *PushTaskOnDeviceQueue()* function. In other words, if the total number of tasks that have been assigned to devices is large enough, then the scheduler will stop dispatching tasks in the ready queue to devices. The intuition is to let the ready queue hold most of the unassigned tasks and sort them in order while guaranteeing that processing units have enough tasks residing in their device queues to be executed. Unlike the greedy dispatching approach that

Algorithm 5 Heterogeneity ratio-based mapping

```
1:  function PUSHTASK(Γ)
2:      Sort tasks in the ready queue by largest-heterogeneity-ratio-first
3:      for t_i in ReadyQueue decreased by H_i do
4:          if H(task) < H(t_i) then
5:              continue
6:          InsertBefore(task, t_i, ReadyQueue)
7:      num ← GetAllDeviceLen()
8:      if num < thr then
9:          PUSHTASKONDEVICEQUEUE
10:
11: function PUSHTASKONDEVICEQUEUE
12:     τ_i^j ← PopFront(ReadyQueue)
13:     for M_q in processor set M do
14:         EFT(τ_i^j, M_q) = T_Avail(M_q) + MDAC(τ_i^j, M_q) + C_{τ_i^j, M_q}
15:     Assign τ_i^j to M_q that minimize EFT(τ_i^j, M_q)
```

assigns ready tasks immediately to devices, our non-greedy approach ensures that tasks entering the ready queue late still have a fairly good chance to be assigned to their favorite processing units. The function *PushTaskOnDeviceQueue* (Lines 15-21) seeks to assign tasks to devices. It first grabs the task with highest heterogeneous ratio (Line 16), then estimates the finish time of this task if assigned to each processor (Lines 17-19), and finally assigns the task to the processor that yields the earliest finish time (Line 20).

Time complexity. This algorithm needs to compute the heterogeneity ratio and do sort insertion that is $O(l^2)$, the assignment phase needs $O(l^2 \cdot m)$ time complexity. The total time complexity is $O(l^2 \cdot m)$ where l is the number of tasks and m is the number of processors.

6.3.2 Kernel Graph Structure Considerations

Our second algorithm improves upon the baseline algorithm by considering the kernel graph structure of each application. As discussed in Sec. 6.2, our observation is that for many applications, the time taken to transfer data among kernels executed on different devices (which heavily depends on

the kernel graph structure) is far from negligible when compared to task execution times. For certain data-intensive applications, the data transfer time is actually the dominant factor in response time performance. Let $T(e_i^{jk})$ represent the general data transfer cost between two dependent tasks t_i^j and t_i^k. Since $T(e_i^{jk})$ can be decided only after knowing the specific devices to which these two tasks are assigned, we compute the average cost as the estimated data transfer time between t_i^j and t_i^k, which is given by

$$T(e_i^{jk}) = \frac{\sum_{q,w \in M} \left(T_{q \to w}(e_i^{jk}) \right)}{m^2}. \tag{6.13}$$

Note that if τ_i^j and τ_i^k are assigned to the same device, then $T(e_i^{jk}) = 0$.

The algorithm seeks to assign higher priorities to tasks with larger $rank(\tau_i^j)$ values. $rank(\tau_i^j)$ is defined as:

$$\begin{aligned} rank(\tau_i^j) &= \sum_{M_q \in M} C_{i,q}^j/m + \max_{\tau_i^k \in \mathcal{S}(\tau_i^j)} \Big(T(e_i^{jk}) \\ &\quad + \sum_{M_q \in M} C_{i,q}^k/m \Big), \end{aligned} \tag{6.14}$$

where $\sum_{M_q \in M} C_{i,q}^j/m$ denotes the average execution time of task τ_i^j, and the $\max()$ term represents the longest time taken to send τ_i^j's data to any of its successor tasks plus this successor's execution time. The intuition behind using $rank(\tau_i^j)$ values is to give pairs of connected kernels that are computation-intensive and/or data-intensive higher possibilities to be assigned to their favorite devices. The pseudo-code of this algorithm is given in Algorithm 6. As seen, the algorithm is identical to our baseline algorithm except that the scheduling priorities tasks using the $rank(\tau_i^j)$ values instead of heterogeneity ratios.

6.3.3 Data Partitioning

According to the observation given in Sec. 6.2, the intuition behind data partitioning is that if a task is data-intensive, then dividing its data into multiple slices would give it a higher chance to utilize more processors. This idea has been proposed and applied in (Lee et al., 2013), but only under

Algorithm 6 Structure rank based heuristics

```
1: function PUSHTASK(task)
2:     rank(task) ← Compute rank of task
3:     for t_i in ReadyQueue decreased by rank(t_i) do
4:         if rank(task) < rank(t_i) then
5:             continue
6:         InsertBefore(task, t_i, ReadyQueue)
7:     num ← GetAllDeviceLen()
8:     if num < thr then
9:         PUSHTASKONDEVICEQUEUE
```

a single kernel scenario. For example, an automated partitioning technique has been proposed in (Luk et al., 2009) to partition the data of a single kernel such that this kernel can be executed on a CPU and a GPU in parallel. Unlike prior work, our third algorithm considers data partitioning as a sub-component and integrates it into our considered multi-kernel scenario.

Despite its advantages, data partitioning may also introduce additional data transfer costs, as discussed in Sec. 6.2. Thus, a mapping algorithm needs to decide whether to apply data partitioning to applications. Our third algorithm extends the baseline heterogeneity ratio-based algorithm by taking data partitioning into account. We apply a historical data profiling technique to decide whether a task needs to be partitioned. In the implementation, we record the historical sampling data and use a non-linear regression-based cost model $(a * D^b + c)$ (National institute for research in computer science and control, 2008) (where a, b, and c are constant coefficients, and D is the data size) to find out the relationship between data size and execution time. Given the data size of a kernel, if the estimated execution time (without applying data partitioning) is larger than a pre-defined threshold, then we partition it into multiple blocks.

6.4 Implementation and Evaluation

In this section, we present the implementation methodology and experimental results used to evaluate the effectiveness of our proposed algorithms.

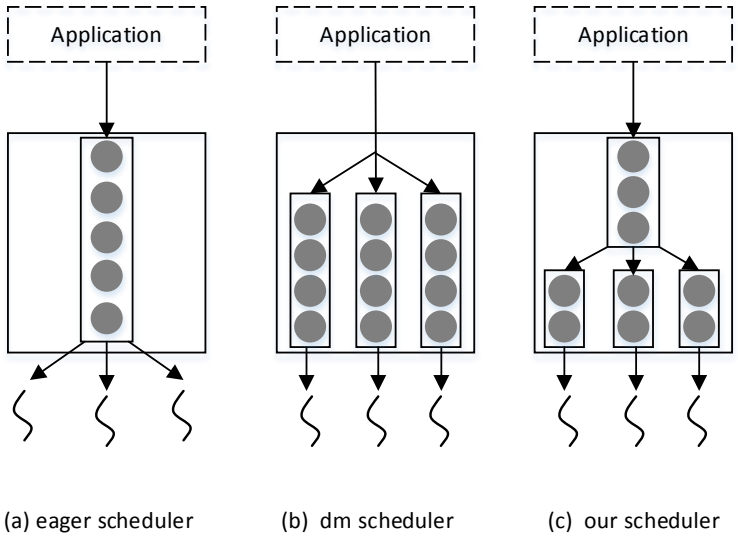


Figure 6.3: Our scheduler implementation

6.4.1 Implementation

We implemented our scheduler algorithms on top of the StarPU runtime platform (Augonnet, Thibault, Namyst, and Wacrenier, Augonnet et al.) as customized schedulers. To better support our algorithms, we modified part of StarPU's core code. The role of the StarPU scheduler is to dispatch tasks onto different processing units (named "workers" internally). All StarPU scheduling strategies implement task dispatching using queue-based method. Tasks that have received needed data from their predecessors are pushed into a $ReadyQueue$. This $ReadyQueue$ is updated at runtime while tasks arrive dynamically. Based upon this dispatching model, our schedulers make mapping decision at runtime for tasks in $ReadyQueue$.

StarPU has several pre-defined schedulers, including the eager scheduler, the dm scheduler, and the dmda scheduler. The eager scheduler uses a single FIFO task queue, as illustrated in Fig. 6.3 (a), from which workers draw tasks to execute. The mapping decision is made only when a worker becomes idle. More complex schedulers such as the dm scheduler maintain one queue for each processing unit, as shown in Fig. 6.3 (b). A task is immediately dispatched to a specific worker once it is pushed into the $ReadyQueue$. Different from these implementation strategies,

our scheduler uses a central priority queue to hold and sort tasks, and dispatch tasks to worker's private queues, as illustrated in Fig. 6.3 (c). Under our implementation, the proposed schedulers do not immediately dispatch an incoming task to one of the workers' queues. Instead, we set a threshold value (as discussed in Sec. 6.3.2) to trigger the dispatching action. The central priority queue would dispatch tasks to workers only when the total number of tasks residing in workers' queues is less than the pre-defined threshold value. A large threshold value may allow the scheduler to have a better ordering of the $ReadyQueue$. However, when considering multiple application scenarios, the total number of tasks could be large. Since pushing tasks into the $ReadyQueue$ may incur overheads, a large threshold value may also reduce the efficiency as such overheads negatively impact the timing performance. Although depending on the specific hardware, the idea behind setting a threshold value is to perform task pushing and task execution in parallel at runtime.

6.4.2 Experimental Setup

We implemented the proposed algorithms in a real heterogeneous desktop computer consisting of a CPU and two discrete GPUs. The hardware specification is given in Fig. 6.4. The benchmarks used in the experiments are listed in Table 6.4. All benchmarks are rewritten in order to be used on the StarPU runtime platform. Among the benchmarks, MonteCarlo and Cholesky factorization are considered to be computation -intensive because they have relatively heavier computing workload for processor units and have a relatively high computation-to-communication ratio (i.e., the kernel execution time is far greater than the time to transfer its needed data from another device). On the other hand, VectorAdd and VectorIncrement are considered to be date-intensive because their computing workload is low, but may generate heavy data traffic. To reflect different workload scenarios, we vary the problem scale of each benchmark to three problem sizes.

The specific values of the problem sizes generated in the experiments are shown in Table 6.4. Moreover, we test three workload composition scenarios commonly seen in practice, i.e.,

	CPU	GPU1	GPU2
Architecture	Intel Core i7-4770	NVIDIA GeForce GTX 660	NVIDIA GeForce GT 620
Frequency	3.9 GHz	1033 MHz	700 MHz
Memory	16GB DDR3	2048MB GDDR5	2048MB DDR3
OS	64-bit Linux Ubuntu lucid		

Figure 6.4: Experimental Hardware Specification

computation-intensive, data-intensive, and randomly mixed workloads. To generate these composition scenarios, we first generate one instance of each of the seven benchmarks shown in Table 6.4 as the base case. We then generate the computation-intensive workload composition using the base case combined with three instances of each of the two computation-intensive benchmarks (mentioned above). Similarly, the data-intensive workload composition is generated using the base case combined with three instances of each of the two data-intensive benchmarks. The mixed workload composition is generated by creating two instances of each of the seven considered benchmarks. Note that the current StarPU runtime system implementation mainly considers the single application scenario. To support simultaneous execution of multiple applications, in our experiments, we compose all the benchmarks into one single executable file by rewriting and compiling the source codes of the benchmarks using StarPU's SDK.

We compare our proposed mapping algorithms against the best available scheduler of StarPU—the dmdar (deque model data aware) scheduler, which considers the task execution time and the data transfer time when making mapping decisions. It is similar to the classical heterogeneous-earliest-finish-time-first scheduling (HEFT): dmdar schedules each task to a processing unit that provides the minimum finish time, and sorts tasks residing in each worker queue by the largest number of available data buffers first. Moreover, we compared our algorithms to the integer programming formulation, which yields an optimal (theoretically) solution. For each experimental setup, we tested two system configurations: one with one CPU and two GPUs, and the other one

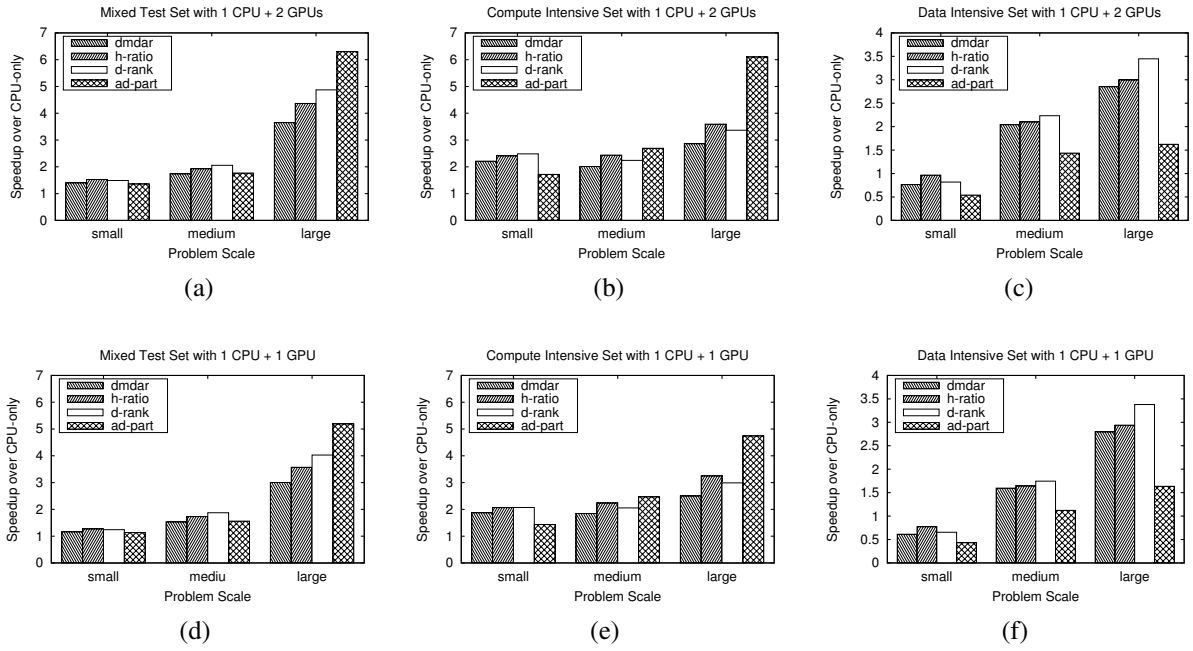


Figure 6.5: Experimental results on the competition time. In all six graphs, the x-axis denotes the three tested scenarios where problem size scale is varied to be small, medium, and large (according to Table 6.4). The y-axis denotes the speedup each algorithm achieved upon the naive CPU-only mapping algorithm. Graphs in the first (second) row depict the results under the system configuration with one CPU and two GPUs (one CPU and one GPU). In the first (respectively, second and third) column of graphs, mixed (respectively, computation-intensive and data-intensive) workloads are assumed.

with one CPU and one GPU (GTX 660). Regarding the evaluation metric, we measured the final completion time for running each entire experiment set. In the following, we denote our baseline mapping algorithm (Sec. 6.3.1), structure-based mapping algorithm (Sec. 6.3.2), data partitioning-based mapping algorithm (Sec. 6.3.3), the dmdar scheduler, and the integer programming solution, as “h-ratio”, “d-rank”, “ad-part”, “dmdar”, and “IP”, respectively.

6.4.3 Results

The obtained experimental results comparing our mapping algorithms against dmdar are shown in Fig. 6.5 (the organization of which is explain in the figure’s caption). Each bar plots the speedup

achieved by the corresponding algorithm upon a naive CPU-only mapping algorithm which prioritizes workloads by shortest-execution-time-first and maps all workloads only to CPU.

As seen, in most tested scenarios, our proposed mapping algorithms improve upon dmdar. The performance gain varies depending on the workload composition and problem scale. As shown in all six graphs of Fig. 6.5, when the problem size is small or medium, one or more of our proposed algorithms yield slightly better performance than dmdar. The improvement is not significant in these cases because the variances in heterogeneity ratio and structure spawn are small. Thus, the benefit of specifically considering these factors becomes less significant. When the problem size becomes large, the performance improvement achieve under our proposed algorithms becomes more substantial. For example, as seen in Fig. 6.5(b), for computation-intensive workloads with large problem size, h-ratio, d-rank, and ad-part improve upon dmdar by more than 15%, 10%, and 110%, respectively. In particular, ad-part achieves the best performance in these cases because computation-intensive kernels are divided into parallel threads with partial data. This effectively reduces the time to complete such kernels when multiple processing units become available. Moreover, for computation-intensive kernels, applying data partitioning does not incur much data transfer penalty. Another interesting observation is that when workloads become data-intensive, ad-part yields the worst performance, as shown in Fig. 6.5(c) and (f). By analyzing the mapping traces of these experiments, we observe that partitioning data-intensive applications may incur significant data transfer time, which negatively impacts the completion time performance. Unlike prior work considering single application scenario where data partitioning should be applied in most cases, our results suggest that data partitioning should only be selectively applied, in particular when workloads become more data-intensive. Fig. 6.5(d)-(f) show the results under the system configuration with the CPU and only one GPU (removing the less powerful GT 620 GPU). Compared to the case where all three processing units are used (shown in Fig. 6.5(a)-(c)), the observation is that the speedup decreases. This is intuitive because less resources are available in this case.

Table 6.3: Comparison against IP.

	Exp. set 1 case study $\times$ 5	Exp. set 2 va+xgemm+inc $\times$ 5	Exp. set 3 xgemm+fblock+pi $\times$ 2
IL	318.52 ms	758.19	7176.68
Ours	330.28 ms	868.61	9975.14

Comparison against IP. We have also conducted experiments to compare our proposed algorithms against IP. Since solving the IP given in Sec. 6.1.2 is quite expensive as we experience in these experiments, we choose to only conduct experiments using a relatively small set of applications. Table 6.3 shows the application set used in the experiments and the results under IP and our algorithm (we select the best result produced under the three algorithms). As seen, our algorithm achieves comparable performance to IP while yielding a much lower runtime complexity.

6.5 Summary

In this chapter, we investigate the problem of mapping multiple applications implemented using kernel graphs in a heterogeneous system consisting of CPUs and GPUs. To achieve fast completion time, we present a fine-grain mapping framework that explores a set of critical factors that are suggested by several measurements-based case studies. We present a theoretical framework that formulates this problem as an integer program and a set of practically efficient mapping algorithms. We implement the proposed algorithms in a real heterogeneous system and conduct extensive experiments using a set of popular benchmarks. Experimental results demonstrate that our proposed algorithms can achieve up to 30% faster completion time compared to the state-of-the-art mapping techniques, and can perform consistently well across different workloads. An interesting future work is to extend the problem space to allow applications to have pre-defined completion time requirements, which is often seen in embedded systems in practice. This would make the problem even more challenging because greedy mapping choices may easily cause applications to miss their timing requirements.

Table 6.4: Benchmarks used in experiments

Benchmark	Description	Small Problem Size	Midium Problem Size	Large Problem Size
xgemm	Combined matrix multiplication and addition	1k*1k matrix x 3	4k*4k matrix	8k*8k matrix
cg	Conjugate Gradient	1k*1k matrix and 1k vector	4k*4k matrix and 4k vector	8k*8k matrix and 8k vector
cholesky	Cholesky matrix factorization	1k*1k matrix	1k*1k matrix	4k*4k matrix
increment	Vector incrementation	10k vector	100k vector	1M vector
va	Vector Add	10k vector	100k vector	1Mk vector
pi	Monte Carlo method to compute pi	1k hits per task, 1k tasks	4k hits per task, 1k tasks	8k hits per task, 1k tasks
fblock	3-D assignment	128*128*128 cube	256*256*256 cube	512*512*512 cube

CHAPTER 7

EXPLORING COMPUTATION AND DATA REDUNDANCY VIA PARTIAL GPU COMPUTING RESULT REUSE [1]

7.1 Case Study

We conduct a measurement-based case study to motivate the potential benefit of result reuse in GPGPU computing. An error-free reuse would leave no room for approximate memoization techniques. This requirement combined with the blackbox nature of GPGPU implies that the entire input data and the entire kernel code should be an exact match for any possibility of reuse. Throughout this chapter, we use a measurement called redundancy to deduce the final possibility of reuse for GPU computations. Redundancy is defined as two equivalent GPU kernel launches with the exact same input data and kernel code.

We use a publicly available dataset movieLens (Harper and Konstan, 2015), which is a collection of rating data from the MovieLens web site. We used two versions of movieLens: one denoted "movieLens-20M" released in 2015 which contains 20 million ratings applied to 27,000 movies by 138,000 users; the other one denoted "movieLens-26M" released in 2017 which contains 26 million ratings applied to 45,000 movies by 270,000 users (Harper and Konstan, 2015). We run a commonly performed matrix factorization function (Koren et al., 2009) on both versions of movieLens. For these datasets, the corresponding matrix is a 27,000 $\times$ 138,000 and a 45,000 $\times$ 270,000 sparse matrix, respectively. The first two rows of Table 7.1 show the results for an unabridged redundancy among these two datasets. Since these two databases are different, the redundancy is 0, implying that there does not exist any pair of GPU computing requests exhibiting exactly the same input data.

[1] ©2018 ACM. Reprinted, with permission, from Husheng Zhou, Soroush Bateni, and Cong Liu. "GRU: Exploring Computation and Data Redundancy via Partial GPU Computing Result Reuse", In Proceedings of the 32nd ACM International Conference on Supercomputing (ICS18). DOI:10.1145/3205289.3205318

Table 7.1: Movie recommendation using GPU-enabled Spark on two movieLens datasets with different partitioning.

Datasets	Block	Dim (K)	Launch	Redundancy	
movieLens-20M	1	1650x288	1	0	0%
movieLens-26M	1	5290x920	1		0%
movieLens-20M	144	11.5x2	156	88	56.41%
movieLens-26M	460	11.5x2	483		18.22%
movieLens-20M	2304	2.8x0.5	2352	2112	89.80%
movieLens-26M	7520	2.8x0.5	7614		27.74%

Next, we set out to test the effectiveness of a straightforward implementation of partial result reuse that divides the input data into smaller chunks. We use a popular big data processing infrastructure–Spark (IBM, 2016) to process movie recommendations. The Spark framework is in charge of running a map-reduce algorithm, in turn partitioning the matrix into smaller blocks to improve parallelism on different workers.

For each dataset, we performed two runs with different partitioning parameters, depicted in the rest of Table 7.1. The first column of Table 7.1 distinguishes the datasets. The second column indicates the number of blocks the original dataset is partitioned into, and the dimension of each block (in Kilo rows/columns per block) is shown in the third column. The fourth column is the number of invoked GPU kernel launches. The last column indicates the redundant GPU computation between the two runs with the same dimension. For example, when we partition datasets into 11.5K $\times$ 2K blocks, the movieLens-20M dataset is partitioned into 144 blocks and invokes 156 GPU kernel launches, while the movieLens-26M dataset is partitioned into 460 blocks invoking 483 kernel launches. Among these two runs, 88 kernel launches are redundant, meaning that the input data and computation are equivalent, accounting for 56.41% and 18.22% of total kernel launches within each run respectively.

Through data partitioning, we observe a good degree of redundancy between the two runs. This is intuitive as movieLens-26M is similar in nature to movieLens-20M, implying that they contain a large fraction of equivalent data blocks. Furthermore, when we further partition these datasets

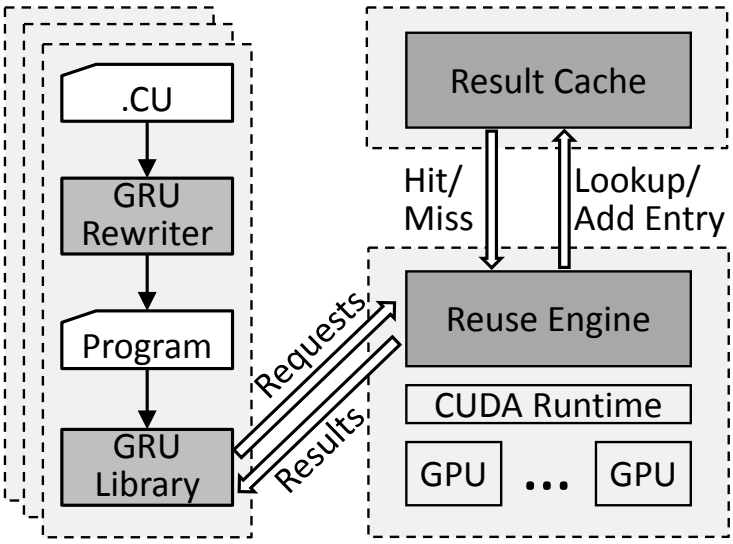


Figure 7.1: GRU architecture consists of a rewriter and a library at the front-end, and a back-end service that runs in the cluster.

into finer-grained 2.8K×0.5K blocks, the redundant computations account for 89.8% and 27.74% among the two runs. This is because with a finer-grained block size, more fractions of the datasets can potentially become equivalent.

This case study reveals a considerable redundancy in production runs on similar or incremental datasets. Such redundancy is expected to be even more significant in large-scale functionality-dedicated clusters/datacenters where thousands of users/applications perform computation requests of common interests (Gunda et al., 2010). This case study also highlights the following critical observation: rather than relying on exact result reuse which may suffer from low redundancy due to coarse-grained consideration of input data, exploring partial result reuse on GPU could potentially elevate the reuse idea to become effective in practical settings due to increased redundancy and reusability.

7.2 GRU Design

7.2.1 Overview

Our core idea behind GRU is to avoid redundant computations on GPU. This is achieved by reusing the results of previous computations that exhibit the same computing code and input data. The architecture of GRU is illustrated in Fig. 7.1. GRU consists of a front-end that runs on the user's desktop or an instance in the cloud, and a back-end service running on a cluster or another instance(s) in the cloud. The front-end is composed of a rewriter and a library which are shown as shadowed rectangles. The back-end consists of a reuse engine and a result cache.

GRU is designed to be transparent to end-users. End-users shall transparently use the GRU rewriter to compile a CUDA source code and run the generated binary executable as a normal CUDA program. At runtime, the GRU library forwards the GPU computing requests including GPU memory allocations and kernel launches to the reuse engine located in the back-end. The results are then retrieved by GRU and returned to the program. The CUDA program can be run on a physical desktop or a virtual instance without a native GPU hardware since all the actual GPU computations are launched on the back-end.

The reuse engine that resides in the GRU back-end is a key component that is in charge of processing incoming GPU computation requests and relaying the output data back to the front-end. The reuse engine uses the hash value of the GPU binary (i.e., .cubin) as the fingerprint of a GPU kernel (i.e., a piece of GPGPU computing code). In addition, the reuse engine uses the hash value of the data chunk as the input's fingerprint. Upon receiving a kernel launch, the reuse engine calculates hash values (of kernel's fingerprint, input's fingerprint, and other primitive type parameters) and consults the result cache. The result cache implemented using an LUT stores previous kernel launch results in the form of metadata. Once the fingerprint is found in the LUT, the actual GPU computations are skipped and the cached result is directly reused. Otherwise, the reuse engine issues the kernel launch request to GPU and relays the output to the front-end. In the meantime, this output is inserted into the result cache as a new entry.

(a) (b) (c)

Figure 7.2: Two similar images (a) (b) with same tile (c).

In our current implementation and evaluation, the reuse engine and the result cache components are located on the same physical machine which is equipped with physical GPUs. However, the result cache can be easily decoupled from the reuse engine and put onto a dedicated cache server, providing cloud/cluster-wide result caching over the network.

7.2.2 Methodology

Reuse Basics

Many GPGPU programs contain a single kernel that is transferred to GPU from main memory. The kernel is then launched on the GPU accompanied by its parameters. GRU treats the result of a single kernel launch as the basic reusable unit. To realize result reuse, GRU needs to first interpret and identify each incoming kernel's code and parameters. If a parameter is of GPU memory pointer type, GRU would need to dereference the pointer in order to obtain the content. The kernel code and the input data can be identified by GRU through intercepting CUDA runtime APIs.

After obtaining the kernel code and input data, GRU dereferences the pointer parameters and obtains the hash value (fingerprint) as the key to probe result cache. If there is a corresponding cached result (cache-hit), it is used as the output of the kernel launch. In the case of a cache-miss, the kernel launch request is forwarded to the CUDA driver for the actual computation on GPU. The computed result along with the fingerprint are then stored in the result cache as a new entry.

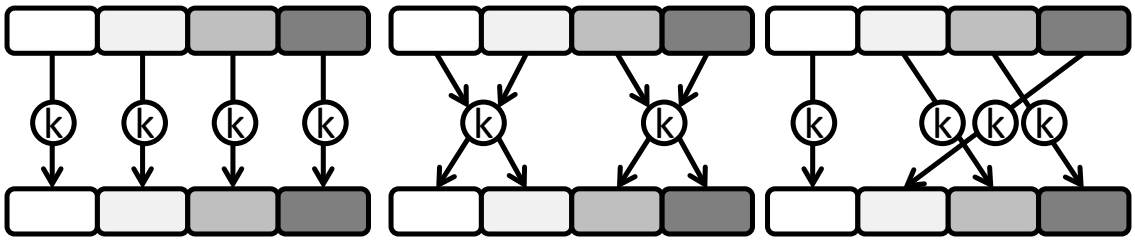


Figure 7.3: Three example data parallel patterns: (a) Map (b) Partition (c) Scatter/Gather.

Partial Result Reuse via Smart Data Partitioning

Under the above-mentioned basic reuse principles, the result of a kernel launch can be reused only if an entry can be found in the result cache that contains the exact same kernel code and parameters. However, for many scenarios in practice, multiple users may execute the same GPGPU kernel program using similar but not exactly the same input data. Fig. 7.2 shows an example scenario, where two similar images (shown in Fig. 7.2(a) and (b)) are being analyzed using the same image convolution kernel. By applying the exact reuse principle, these two launches are not reusable with respect to each other. However, an interesting observation is that there exists a common tile between these two images, as is shown in Fig. 7.2(c). Thus, the result of the same kernel launch using this tile as the input data becomes reusable.

Similar to content-based data partitioning applied in deduplication systems in the context of storage and incremental computation frameworks (Bhatotia et al., 2012, 2011), our intuitive idea is to allow GRU to selectively pre-partition a request's input data into smaller chunks, thus increasing the redundancy and reusability through partial result reuse. GRU is designed to realize the idea of partial reuse by analyzing the kernel code provided by the user and transparently performing a compiler-assisted transformation. The transformation is responsible for converting a normal GPU program into one that can communicate with GRU by injecting GRU API calls wherever deemed necessary. A challenge herein is to determine whether an input data can be correctly partitioned and how to efficiently perform the partition. To resolve this challenge, GRU analyzes general data

```
1  TYPE* a = (TYPE*)malloc(sizeof(TYPE));
2  TYPE* b = (TYPE*)malloc(sizeof(TYPE));
3  TYPE* c = (TYPE*)malloc(sizeof(TYPE));
4  init_data(); // init value of a, b, c
5  struct matrix_handle_t A_handle, B_handle, C_handle;
6  matrix_partition (&A_handle, a, sizeof (TYPE),
7                    slice_a,                  // num of slices
8                    xdim_a, ydim_a,           // dimension
9                    vertical_filter);         // filter function
10 matrix_partition (&B_handle, b, sizeof (TYPE), slice_b,
11                   xdim_b, ydim_b, horizontal _filter);
12 matrix_map_filter (&C_handle, c, sizeof (TYPE),
13                      vertical_filter, slice_a,
14                      horizontal_filter, slice_b);
15 gru_launch (&kernel, A_handle, B_handle, C_handle, …)
   … …
```

Figure 7.4: Code segment of matrix multiplication program after transformation.

parallel patterns that are reliable for the reuse purpose, and is capable of recognizing such reusable data parallel patterns of incoming requests at runtime.

According to McCool and Samadi (McCool et al., 2012; Samadi et al., 2014), data parallel patterns can be generally categorized into six types: map, partition, scatter/gather, reduction, scan, and stencil (see (McCool et al., 2012; Samadi et al., 2014) for detailed definitions of these patterns). A key observation is that applications containing the map and partition patterns (as illustrated in Fig. 7.3(a) and (b)) are well suited for input data partitioning and consequently, for partial result reuse purposes. Intuitively, this is because under these two patterns, the elements of the output can be separated into groups that are independent of each other. As for application exhibiting other data parallel patterns, it is not suitable to perform data partitioning due to rather complicated dependencies. For example, for the scatter/gather pattern as illustrated in Fig. 7.2(c), every element in the input array is randomly accessed by the kernel to produce an output element, making data partitioning unsuitable.

This limitation does not impair GRU's ability to be effective in most real scenarios since a large portion of parallel tasks, applications that use map/reduce processing, exhibit the patterns of map

and partition (Ghazal et al., 2013; Wang et al., 2014; Huang et al., 2010). However, the remaining four patterns are not entirely dismissed by GRU. Even if an application contains scather/gather, reduction, scan, and stencil, there is still a chance for the output to be reused if the output is transcribed using a map or partition pattern. For example, the kernel of a matrix multiplication combines a row and a column of elements of the input matrices into a single element for the output matrix. This operation exhibits the reduction pattern that has inherent dependencies. However, the output is written as independent non-overlapping sections spanning one node which is the partition pattern (McCool et al., 2012). Thus, the input data of a matrix multiplication kernel can be safely partitioned using a row or a column as the smallest partition unit since this scheme of data partitioning will not interfere with the reduction. In other words, it is safe to break up the computations (through data partitioning) that exhibit the partition pattern because these computations are independent. However, breaking up the computations of a reduction pattern will result in an invalid state.

GRU introduces a compiler-assisted approach to identify kernels that allow for data partitioning and performs the proper partitioning at compile time. We setup an LLVM (Lattner and Adve, 2004; Wu et al., 2016) pass to analyze the IR (intermediate representation) of both CUDA kernel code and CPU code. This will determine the access patterns of the input and output data in the kernel. GRU associates the memory access of each memory object with the thread's built-in parameters, such as blockIdx and threadIdx whose value ranges can be inferred according to the dimensions of each kernel. Then, these parameters and their ranges are converted into STP-recognizable constraints and sent to the STP solver (STP, 2014). The STP solver decides whether or not one individual element (map pattern) or a fixed set of elements (partition pattern) are accessed by every thread exclusively. More precisely, GRU can find out if the fixed set of elements are in vertical, horizontal, or block manner. Thus, we can get the partition plan accordingly. Specifically, a kernel is determined to be feasible for partitioning if it contains the aforementioned data access patterns. The compiler pass will then insert a set of GRU APIs that perform the data partitioning using a

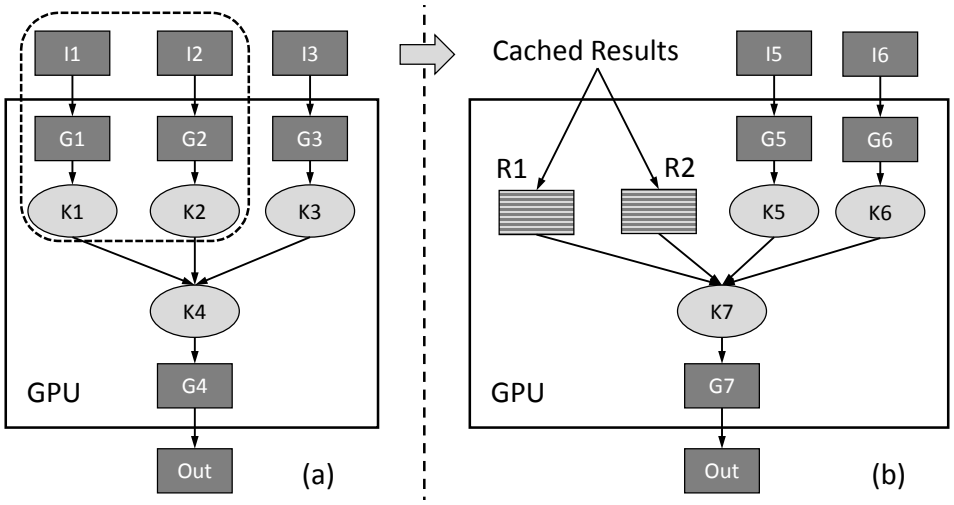


Figure 7.5: Different functionalities share common sub-computation K1 and K2 that can be reused from previous cached results.

set of predefined partition filters: vertical_filter, horizontal_filter, and block_filter for up to three dimensional data arrays. Fig. 7.4 illustrates an example, showing the code segment of the matrix multiplication kernel after transformation (before the transformation, the code only consists of a kernel launch). We note that this code segment is only used for demonstration purposes. In reality, the GRU rewriter performs the code transformation using the LLVM IR and sends the IR to the LLVM backend to generate the machine code directly. As is evident in the figure, lines 5 - line 11 convert the original input data chunks into partitionable data structures. Lines 12 - 14 partition the output data chunks according to the filters applied to the input matrices. Actual data movement and kernel launch happens when gru_launch is called at line 15. The detailed analysis and rewriting algorithm will be discussed in Sec. 7.3.1.

Partial Result Reuse via Sub-Functionality

While we have covered partial reuse given the same computation code so far, it is actually possible to partially reuse results from different input data and different computation codes. The constraint is the fact that some sub-component of the computational flow is shared across two executions, along with a partially reusable input data.

GRU is designed to consider GPU kernel dependency graphs and reuse parts of the execution flow accordingly. Fig. 7.5 illustrates such a scenario where the entire computation serves a single purpose and is depicted in the black square. However, on the inside, the computation consists of multiple sub-kernels. For Fig. 7.5(a), the kernels are K1, K2, K3, and K4. For Fig. 7.5(b), the kernels are K1, K2, K5, K6, and K7. Fig. 7.5(b) depicts the scenario in which the computing results of K1 and K2 are previously cached by GRU. Thus, their computations can be skipped by directly using the cached results and copying them to the GPU device memory to be used as K7's inputs.

7.2.3 GRU Front-End

Rewriter

As was mentioned above, one major role of the rewriter is to transform the original program into a partition-enabled one. Beyond this capability, the rewriter is also in charge of: 1) Identifying the inputs and outputs of each GPU kernel; and 2) Deciding whether to rewrite the vanilla GPU kernel launch as a reuse-enabled kernel launch. We design the Rewriter to be transparent to the user. We employ an intermediate compile-time technique detailed in Sec. 7.3.1. The major role of Rewriter is to intervene at compile time and redirect API calls to the back end through the front-end library. We describe this library and the back-end next.

Front-end Library

GRU front-end library resides in the user-side which is a substitute for the CUDA runtime and computation libraries (e.g., libcublas.so). Front-end library intercepts all CUDA calls made by applications, and forwards each request to the BE. It is also in charge of managing the connection between the user's desktop/instance and BE. After being loaded by CUDA applications, the front-end library automatically establishes a connection with BE that is specified through an environment

variable. For each CUDA call, FE sends the request together with the related parameters to BE, and awaits the response.

Moreover, high performance computing applications often require massive data transfers between the CPU main memory and GPU device memory. In GRU, since all interactions between CPU main memory and GPU device memory occur in backend, such massive data does not need to be transferred between FE and BE. The GPU device memory pointers are never de-referenced on the user-side. Rather, the actual GPU device memory pointer is transferred between the guest and the host in the form of a hex value. Only if a guest memory pointer takes part in a computation on the host the memory chunk will be transferred over.

7.2.4 GRU Back-End

The GRU Back-end is located in the cluster/cloud, and is responsible for executing CUDA calls received from the FE and returning the computed results. For each application in FE, a new process is created on the host to run all the CUDA requests related to that application using an independent GPU context. BE then forwards the requests received from the FE to the CUDA device driver running on the host physical machine. Finally, BE transfers the computed outputs back to FE at user-side.

The Reuse Engine is implemented in the BE as a core component to realize our idea of result reuse, and acts as a bridge between BE and the CUDA runtime libraries. The reuse engine selectively caches GPU computation results and maintains them in a result cache. We use the Sqlite3 in-memory database to store the cache results. Sqlite3 is open-source and widely used in practice. Moreover, Sqlite3's reliability and performance can be guaranteed (Hipp, 2018). We choose an in-memory database mainly due to performance concerns, since the LUT would be retrieved and updated frequently, at least once for every kernel launch request.

The latency of hashing operations is rather small, even with increased data sizes. We use an extremely fast non-cryptographic hash algorithm to perform the hashing (Y.C., 2012), the speed of

which can reach 13.8 GB/s on a 64 bits OS. The possibility of collision is also guaranteed by the hash algorithm (1 in $\sim 1.8 \times 10^{19}$). The latency of the search and update operations is less than 0.05 ms which is trivial compared to a typical kernel execution time (typically $>$ 100 ms). The dump operation is occurred only when a cache-hit occurs. Thus, compared to the actual kernel execution time, time-saving is still achieved. On the other hand, the result insertion operation is rather time-consuming, which also in turn increases with the result size. However, such operation does not affect the user application's response time since it runs asynchronously with the computing results transfer from reuse engine to FE.

By intercepting the incoming requests from FE, the reuse engine decides whether to issue requests to the CUDA runtime or directly reuse cached results. In our current implementation and evaluation environment, the reuse engine is located in the host machine which is equipped with physical GPUs. However, the fact is that the reuse engine can be easily deployed on a dedicated physical server in the cloud. The only requirement is that this machine should contain a sufficiently large RAM for the in-memory database. This machine would serve as a mem-cache server and would provide a cloud-wide result caching and reusing over the network. The detailed workflow of reuse engine is described next in Sec. 7.3.2.

7.3 Implementation Details

7.3.1 Rewriting Algorithm

Our implementation of the Rewriter is deeply embedded in the llvm compiler. The main task for Rewriter is to intercept IRs generated for both CPU and GPU codes. It will then run a separate pass to do a few analysis all at once. First, the Rewriter identifies any variable initialized in the CPU code. It will discard the ones that are not in any shape or form used in the GPU computation. For example, if a variable is allocated by cudaMalloc, then it is considered to be connected to GPU code. The Rewriter will also analyze the GPU code in order to determine the data pattern that exists in the GPU kernels and it can build the dependency graph of data objects.

Algorithm 7 Rewriting Algorithm

```
Require: gridDim      ▷ Dimension of grid
Require: blockDim     ▷ Dimension of thread block
Require: kernel       ▷ GPU kernel to be analyzed
 1: function PARTITIONPLAN(kernel, gridDim, blockDim)
 2:     para[]=list of pointer parameters of kernel
 3:     plan[]=return value
 4:     for p in para[] do
 5:         idx[] ← ∅ indexes of element access
 6:         for ins in use_chain(p) do
 7:             i ← associateBuildin(operand(ins), gridDim, blockDim)
 8:             idx.push(i)
 9:             g.push( pathConstrant(ins))
10:         if stpCheck(g, idx) is OVERLAP then
11:             plan[p]←NONE
12:         else
13:             if typeOf(idx) is VERTICAL then
14:                 plan[p]←VERTICAL
15:             else if typeOf(idx) is HORIZONTAL then
16:                 plan[p]←HORIZONTAL
17:             else if typeOf(idx) is BLOCK then
18:                 plan[p]←BLOCK
19:             else
20:                 plan[p]←NONE
21:     return plan[]
```

The pseudo-code of this analysis is depicted in Alg. 7. The function PartitionPlan will iterate through the parameter list in lines 4-21. Lines 6-8 will get all the element access indexes for each parameter p. Line 9 will get the path condition required to reach the current instruction. Finally, lines 10-21 will decide the final pattern and directly choose a partition plan. For instance, the condition at line 10 will check for any possible overlap of parallel access pattern. If an overlap exists, Rewriter would conservatively discard the kernel as non-partitionable. The remainder of conditions in lines 14, 16 and 18 will check for our three data filters: vertical, horizontal and block filters. PartitionPlan will return a plan that will be used to take an informed action in later stages.

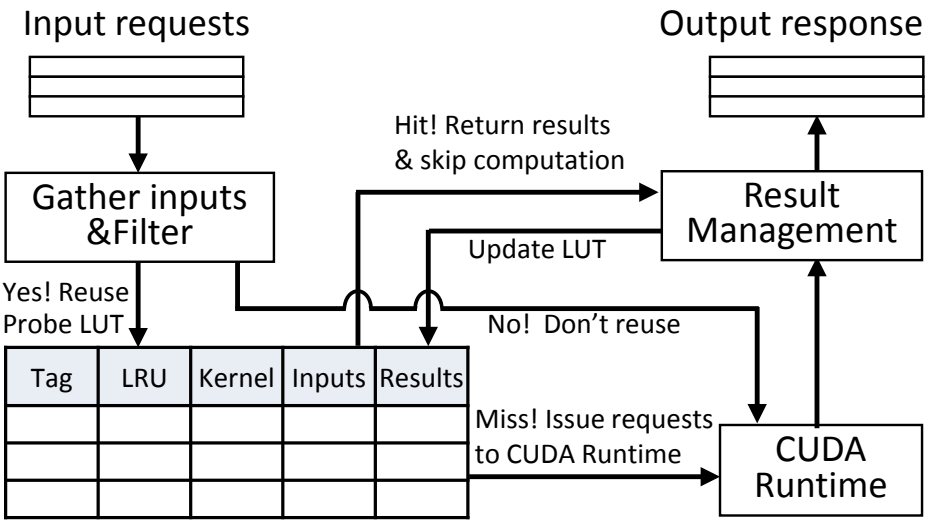


Figure 7.6: Workflow of reuse engine

7.3.2 Result Cache and Reuse

The reuse engine is a key component in GRU BE, which receives kernel launch requests, checks reuse possibility, and returns the cached results or forwards requests to CUDA runtime for actual execution on GPUs. Fig. 7.6 depicts a detailed block diagram of the various components implemented in the reuse engine and shows the general operational flow. Specifically, a kernel launch request is first identified by the reuse engine through its input arguments and data chunk hash values. Then, the launch request is filtered by the reuse engine according to its input size which is directly related to its computation time. We note that a GPU kernel may contain complex data structures as its parameters, which is hard to generate the fingerprint for. These parameters will not be considered for reusing purposes because such complex cases are rare (NVIDIA, 2011).

The filtering process is important because reusing results for small kernels or kernels with a small input data size incurs a considerable overhead, which may be more time consuming than actually executing such kernels on GPU. For functions in libraries provided by the CUDA development kit, such as cublas and cufft libraries, we pre-define the reuse threshold through profiling. The guideline behind finding such threshold values is that reusing results shall yield less execution time than actually launching a kernel on GPUs. For example, we define the reuse threshold for

cublasSgemm[2] to be 4MB, which implies that if the size summation of cublasSgemm's first two matrices is less than 4MB, we will not reuse the result.

If the launch request together with its inputs are worth reusing, then the reuse engine probes LUT to check if there are cached results; otherwise the launch request is directly sent to the CUDA runtime for actual computation. Each entry of the LUT contains the cached results and hash values of kernel code and input arguments. For any pointer argument, the hash value is the hash of its dereference. The hash value for a primitive type argument is itself. There is no semantic loss for arguments between guest and host, since the guest and host communicate through FE and BE which preserve the type information.

A hit occurs in the LUT when there is a cached result, in which case the reuse engine skips the actual computation on GPUs and directly dumps out the cached results to the result manager. When the result manager receives the GPU request for transferring back the result, it performs the actual result transfer. After an actual CUDA kernel launch, the results are supposed to be stored in the GPU device memory. There are three possibilities for such results depending on application scenarios: (*i*) results are transferred to host memory, (*ii*) results stay in the device memory for further computations, and (*iii*) results are copied to another location on the same GPU device memory. The result manager operates differently according to these three scenarios. Scenario (*i*) is the most common and straightforward. The result manager simply copies the cached results directly to the target host address. For scenarios (*ii*) and (*iii*), the cached results need to be transferred to the GPU device memory through additional CUDA calls for further computations, since the data on GPU is out-of-date (in (*ii*)) or invalid (in (*iii*)).

On the contrary, a miss in LUT indicates that result of the current computation request is not available for reuse. The computation request is then issued to the physical GPUs. A new entry is reserved for this computation in LUT. After the actual GPU computation completes, the corresponding entry will be updated accordingly.

[2]cublasSgemm is a CUDA linear algebra function in the CUBLAS library that perform matrix operations. It performs $A \times B + C$ where A, B and C are its three matrix inputs

If the reuse engine is deployed in a dedicated cache server, then cloud-wide GPU computing results can be pooled in such a server over the network, which increases the number of cached results and thus the reusability.

7.3.3 Global object tracking

We introduce the global object tracking optimization to track the content of GPU memory objects, in order to eliminate redundant data hashing of the same GPU memory object. For a majority of GPGPU computation requests, their parameters are in the form of GPU device memory pointers. In order to get the hash value of the kernel arguments, we need to first obtain the contents of the corresponding GPU memory objects. A clumsy method is to dump out the content of GPU objects, then perform the hash. However, under this method, for large objects, the GPU-to-CPU memory transfer may be quite time-consuming.

To eliminate such redundant hashing, GRU maintains a GOM (Gloabl Object Map) to track the content of GPU objects, which is inspired by Moxie–a distributed dataflow engine for GPU object passing (Rossbach et al., 2013). Each entry in GOM records a GPU object's handler, its corresponding CPU object, size, an up-to-date flag, shape, and hash value. An entry of a GPU object is registered to GOM when calling cudaMalloc(), and removed when calling cudaFree(). GRU uses GOM to keep the internal consistency of memory objects. If any CUDA request parameters are passed by the reference (typically a pointer), GRU can easily track the content of the GPU objects. Thus, when a GPU memory pointer is used for multiple kernel launches as the input parameter, it needs only to be hashed once. Besides, through GOM, we can easily identify a pointer which points to an arbitrary address within the memory space allocated to this object, according to the base address and size of a GPU memory object recorded in GOM. Redundant hashing can thus be further eliminated.

7.3.4 Delay transfer

The input data transferred through the network from the front-end to the back-end is unnecessary if a kernel launch can reuse the cached results. In a case like that, the input data is only used for hashing but is not needed to perform the actual computations. Consequently, for a large data requested by a *cudaMemcpyHostToDevice* call, we delay the actual transmission until the corresponding computation is launched, similar to the optimization in PTask (Rossbach et al., 2011).

If a computation request together with the hash value of the kernel code pass the reuse filter, the FE would first hash the input data chunk and transfers the hash value to BE instead of the entire data. If there is a cached result, then the actual input data does not need to be transferred. This delay operation is advantageous particularly for data-intensive applications.

7.4 Evaluation

In this section, we first evaluate the performance gain when deploying GRU on Apache Spark and then evaluate GRU's performance breakdown and incurred runtime overhead using a set of popular micro-benchmarks.

7.4.1 Experimental Setup

In our evaluations, we use two different setups for Spark use cases and micro-benchmarks. For Spark experiment, we adopt GRU to SparkGPU (IBM, 2016) in a five-node cluster that consists of five AWS instances (AMAZON, 2006). The cluster includes four CPU-only t2.xlarge instances and one GPU-powered g3.8xlarge instance that equip with two NVIDIA Tesla K80 GPUs. The GRU frontend resides on four CPU-only nodes and backend resides on the GPU node. When user launches a Spark program, four CPU nodes are used as workers, while the GPU computations are forwarded to the GPU node. For micro benchmarks, we adopt GRU on a desktop that are equipped with an i7-4790K 4.0 GHz processor, 128 GB memory and an NVIDIA Quadro 6000

GPU. To simulate the multi-tenants scenario, we use Xen hypervisor (Barham et al., 2003) to virtualize two instances (DomainU) as tenants. The GRU frontend residents in the instances, and the GRU backend is installed in Domain0. Benchmarks execute inside the instances, and the GPU computations are forwarded to and processed at the host machine (Domain0).

7.4.2 Spark Use Cases

We adopt GRU to SparkGPU (IBM, 2016) in a five-node cluster which includes four t2.xlarge instances and one g3.8xlarge instance. We conducted two sets of experiments. The first one investigates the benefit of applying GRU when different functionalities related to text mining are incurred upon the same dataset, while the second experiment studies the scenario where a commonly performed log analysis functionality is performed on different log traces.

Reuse with Different Functionalities

In this experiment we run four programs – WordStatistics, FreqWord, MostProduct, TopRatio on the public amazon review dataset (McAuley et al., 2015). WordStatistics records the number of each word's occurrences and the number of products that it appears in. FreqWord finds out the top K most frequently used words in the entire dataset. MostProduct looks for the top K words that appears in each category of products. TopRatio gets the ratio of the top K mostly used words among all words. We use two versions of the dataset: a compact version with a size of nearly 8GB and a full version with a size of nearly 20GB. For each dataset, we enable GRU and run the WordStatistics program to cache the results. And the subsequent three programs can benefit from reusing cached results due to the WordStatistics program.

We measure the turnaround time (TAT) which indicates the time elapse from program launch till its completion, and GPU occupancy time (GOT) that is the accumulated GPU computation time on every GPU device. Reducing GOT is crucial when GPUs are rare computing resources in a cluster, since GPUs can be yielded for other computing requests. As shown in Fig. 7.7, the x-axis

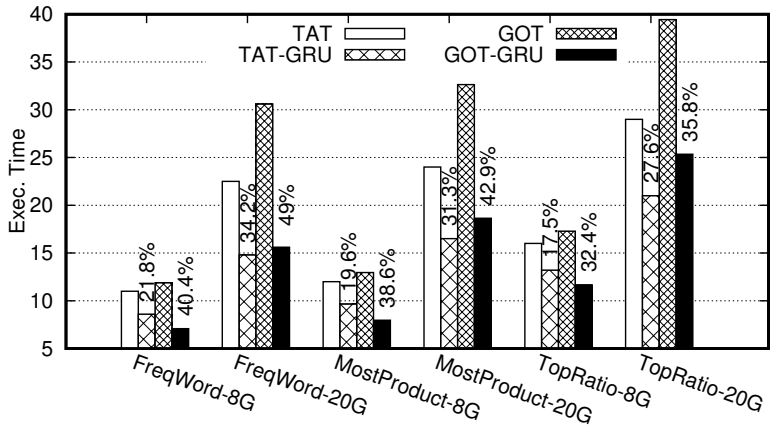


Figure 7.7: Turnaround time (TAT) and GPU occupancy time (GOT) of three programs on two datasets with GRU off and on.

represents three programs on two datasets. The y-axis is the execution time in minutes. The four histograms represent TAT without and with GRU on, and GOT without and with GRU on. The labels on top of the histograms indicate the percentage of the saved TAT and GOT. We observe that all three programs benefit from GRU even if they incur different functionalities, reaching an average of 25% TOT reduction and 39% GOT reduction. Furthermore, with a larger dataset (20GB version vs. 8GB version), the percentage of saved TOT and GOT becomes larger. For example, FreqWord-8G has 21% TAT reduction and 40% GOT reduction, while FreqWord-8G reaches 34% and 49%, respectively. This is intuitive as heavier workloads may exhibit higher degrees of redundancy, and thus, reusability.

Reuse with Different Datasets

In this experiment, we evaluate the case where the same functionalities are performed using different datasets. We considers herein a log tracing system on an extremely popular big-data processing framework, Apache Hadoop (Lab, 2015). We use publicly available data traces of the CMU OpenCloud Hadoop cluster from January of 2011 until June 2011 (Lab, 2015). We dedicate this experiment to an accumulative data trace analysis that is assumed to be executed monthly. This analysis is run to generate statistical data of each job's execution time, summed for the year 2011.

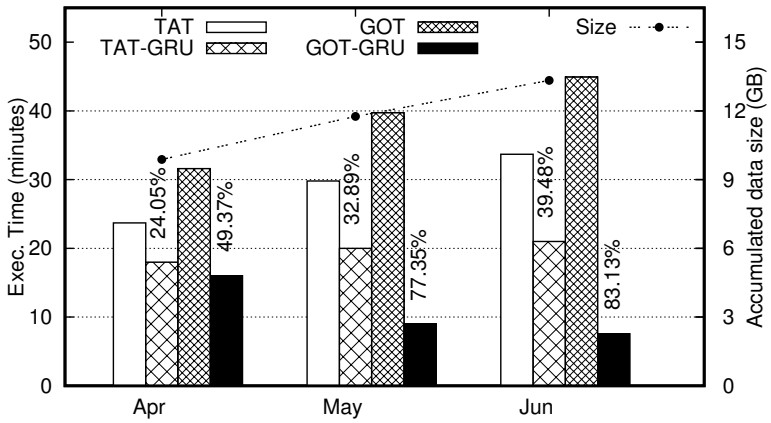


Figure 7.8: Cumulative turnaround time and GPU occupancy time savings for opencloud trace dataset.

While jobs rarely span between months, it is nonetheless necessary to include the accumulative results for administrative purposes.

Fig. 7.8 shows the results of running this scenario for the months of January until June, while we limit the exhibition to three months for clarity purposes. As is evident from the figure, while the database continues to grow (as is shown using connected dots) from 9.8 GB to 13.3 GB, TAT under GRU continues to be consistent, not exhibiting much increase due to the benefit of result reuse.

Another counter-intuitive observation is that GOT dramatically drops after each month. This is because GRU eliminates the need of recomputing the log data from previous months, but rather directly reusing results from previous computations. This trend of reduction in GOT is consistently observed from January until June. Fig. 7.8 shows the total accumulative GOT savings until the month of June, which yields an 83% reduction.

Reuse with Different Functionalities and Datasets

In this section, we measure the efficacy of GRU under a mixed reuse scenario. A mixed scenario is when both the input data and the computation kernels are different between two runs, as discussed in Sec. 7.2.2. Table 7.2 shows an example execution under a potential mixed scenario. The first

Table 7.2: Reuse of non-identical data & computation.

Warmup	Main Run	TAT (% saved)	GOT (% saved)	Reuse
WS-8G	FW-20G	19.9m (11.56%)	25.2m (17.38%)	4.99%
FW-20G	WS-8G	10.1m (21.09%)	6.4m (45.3%)	12.50%

row of Table 7.2 depicts an example run of FreqWord on Amazon Reviews 20GB (Full). However, prior to its execution, a warmup run of WordStatistics is executed using the Amazon Review 8GB (Compact). As is evident in the table, the reuse possibility under this scenario is noticeable but not significant.

On the other hand, the second row of Table 7.2 shows a reversed scenario, in which FreqWord on Amazon Reviews 20GB is used as the warmup execution. For the main execution of WordStatistics on Amazon Review 8GB, the savings are quite significant. In the second scenario, the fact that Amazon Review 20GB is an incremental version of Amazon Review 8GB has contributed a significant boost to the execution time of WordStatistics, saving 45% GPU occupancy time in total.

7.4.3 Experiments with Micro-benchmarks

Setup

The evaluated benchmarks are selected from various application domains and are run in the desktop environment mentioned in Sec. 7.4.1. These benchmarks include a range applications from the Rodinia test suits (Che et al., 2009), CUDA SDK examples (NVIDIA, 2015), a synthesized CUDA game, an object recognition application as listed in Table 7.3. By increasing the application variety, our goal is to perform a reasonably comprehensive evaluation study and identify GRU's strengths and limitations when being applied to applications with different characteristics.

The baseline used in our experiments is rCUDA (Duato et al., 2010), which is a popular API-remoting-based GPU virtualization framework. rCUDA uses the TCP/IP network stack to communicate between its FE and BE. The overhead introduced by the architecture of API-remoting has

Table 7.3: Evaluated benchmarks

chess (Chess game AI)	bfs (breadth first search)
lud (LU decomposition)	mmult (matrix multiplication)
nn (nearest neighbor)	bs (BlackScholes modeling)
hs (hotspot simulation)	decode (Image decoding)
hw (medical imaging)	bp (back propagation)
srad (image processing)	hog (Object recognition)

been extensively discussed in rCUDA (Duato et al., 2010). Thus, in our experiments, we mainly compare GRU with rCUDA to evaluate the efficacy of GPU result reuse, since GRU is almost identical to rCUDA if its result reuse component is disabled.

To evaluate the overhead incurred under GRU, we evaluate two settings: (*i*) we run each experiment under GRU with an empty result cache. Thus, in this setting, the measured performance penalty can be viewed as purely the overhead incurred by GRU. We denote this setting by "GRU-miss." (*ii*) we run each experiment under GRU assuming that the same experiment has been performed by another VM under GRU (i.e., cached entries have already been established in the result cache). Thus, the measured performance in this setting can reflect the effectiveness of GRU. (Note that GRU can only be effective if some reusable results have been cached.) We denote this setting by "GRU-hit". The main performance metric we adopt in our evaluation is the execution time under rCUDA versus GRU (i.e., from the time when the CUDA context is created to the time when the context is destroyed), as this metric directly reflects the effectiveness of the core GPU result reuse idea implemented.

Results

We have also assessed GRU's performance and overheads using a set of popular micro-benchmarks, which are shown in Fig. 7.9. We observe from Fig. 7.9 that most benchmarks (9 out of 11) reach more than 1.25x speedup compared to rCUDA. Benchmark mmult yields the highest speedup (5x) because of its compute-intensive nature (which can be seen in 7.10). An important message received from this set of experiments is that GRU is most effective when applied

to compute-intensive applications. This is because the computation time, which is the most time-consuming component of such applications, can be saved by directly reusing the cached results. On the other hand, GRU is also quite effective in handling data-intensive applications. For example, the bfs benchmark, which is data-intensive as seen in Fig. 7.10, gets a nearly 2.5x speedup. This is mainly because GRU can only optimize the execution of data-intensive applications by merely transferring the hashed value of input data instead of the entire data chunks. Thus, once the cached results associated with the same hashed value are retrieved in LUT, the time-consuming transfer time of input data can be skipped. Moreover, by jointly considering the overhead that GRU introduces when handling these two types of applications, the observation is that GRU performs extremely well for compute-intensive applications due to (*i*) very small overhead incurred when under GRU-miss, and (*ii*) large saving of execution time under GRU-hit; while for data-intensive applications, GRU can still achieve a considerable amount of saving of execution time under GRU-hit but at the cost of more overhead under GRU-miss due to large input data hashing.

On the other hand, we observe that one specific kind of application may not benefit a lot from GPU result reuse, i.e., applications that are not compute-intensive, but have rather small input data and large output. This is because in the case of cache-missing, they introduce non-negligible overhead due to inserting the large output in LUT, while in the case of cache-hitting, they cannot gain much performance speedup due to the non-compute-intensive nature. For example, the decode application yields the lowest performance speedup, because the introduced overhead of result insertion and dumping neutralizes the time saving due to the skipped input data transfer and kernel launch time.

We evaluate GRU's runtime overhead compared to rCUDA (which is mainly due to LUT-related operations) by demonstrating the detailed breakdown of applications' execution times on GPU. Fig. 7.9 depicts the normalized execution time when executing each application under three settings: rCUDA, GRU-miss and GRU-hit. The x-axis of Fig. 7.9 represents each evaluated application, and the y-axis represents the execution time normalized with respect to the execution

time yielded under rCUDA. We observe that the overheads incurred by reuse-miss are less than 5% for four applications, less than 7% for another three applications. Overall, for a majority of the considered benchmarks (7 out of 11), the overhead causes a less than 7% increase in execution times. Among all applications, lud, bfs, and chess yield the greatest overhead (14%-18%). This is because they are either data-intensive or process a large input data, causing more overhead in data hashing, insertion, and movement.

To clearly understand the overhead sources, we have also recorded the detailed breakdown of each benchmark's execution time on GPUs under three scenarios, as shown in Fig. 7.10. There are in total seven components that contribute to an application's execution time: (*i*) Init indicates the initialization time of CUDA context, (*ii*) H2D and D2H indicate data movement between host memory and GPU device memory, (*iii*) Launch indicates the kernel launch time on GPU, (*iv*) H+R indicates the time spent on data hashing and probing in LUT, (*v*) Insert indicates the time spent on inserting results in LUT when caching a result, (*vi*) Reuse means a kernel launch is skipped by reusing cached results. Through examining the breakdown, we can clearly figure out different characteristics of benchmarks and the overhead differences between GRU-miss and GRU-hit. For the bfs benchmark that incurs the most observable overhead, we can see in Fig. 7.10 that compared to other benchmarks, the H2D, D2H, and H+R components under bfs contribute a larger portion of the total execution time. The data transfer time in bfs accounts for 81% (67% for H2D and 14% for D2H) of the total execution time under rCUDA, 65% under GRU-miss, and only 20% under GRU-hit. With the delay transfer optimization implemented in GRU, the H2D time of bfs can almost be eliminated under GRU-hit.

7.5 Related Work

Managing GPUs in the cloud. Current approaches for GPU management in the cloud are classified into I/O pass-through (AMAZON, 2006), API-remoting (Duato et al., 2010; Giunta et al., 2010; Lagar-Cavilla et al., 2007; Shi et al., 2012), para-virtualization (Dowty and Sugerman, 2009;

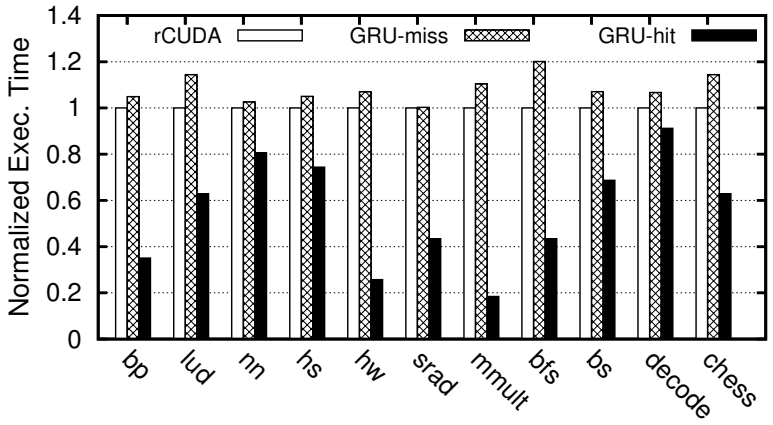


Figure 7.9: Performance with respect to normalized execution time.

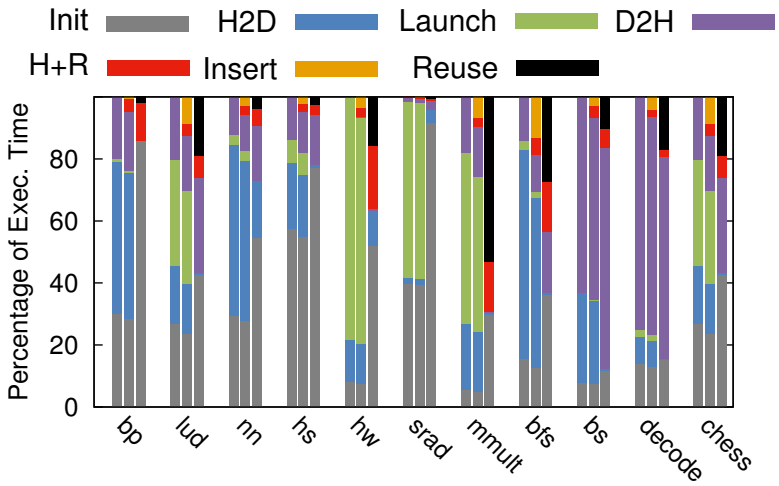


Figure 7.10: The three histograms for each benchmark represent the breakdown under rCUDA, GRU-miss and GRU-hit, respectively.

Gottschlag et al., 2013; Suzuki et al., 2014) and full-virtualization (Suzuki et al., 2014; Tian et al., 2014; Malka et al., 2015; Zhou et al., 2015), the latter two being two different implementations of the device emulation technique. However, these works do not exploit the idea of GPU computing result reuse.

Computing result reuse. The concept of CPU-based computation reuse has been proposed in the programming language and computer architecture communities. Compiler-assisted approaches (Sodani and Sohi, 1997; Conners and Hwu, 1999; Connors et al., 2000; Ding and Li, 2004) seek to reuse intermediate results at CPU instruction level. Function-level memo-

ization (Michie, 1968; Pugh and Teitelbaum, 1989) is used to avoid re-executing functions by caching the results of prior function calls. Moreover, frameworks are proposed to reuse redundant computations at a higher level for the emerging incremental data processing field. For example, Spark (Zaharia et al., 2010), Percolator (Peng and Dabek, 2010), and CBP (Logothetis et al., 2010) provide programmers with facilities to store and reuse states across successive runs; while Dryad-Inc (Popa et al., 2009), Nectar (Gunda et al., 2010), Haloop (Bu et al., 2010), Incoop (Bhatotia et al., 2011), CIEL (Murray et al., 2011), and Shredder (Bhatotia et al., 2012) are systems that reuse prior computing results. On GPU-incurred reuse, Arnau et al. (Arnau et al., 2014) presented a hardware memoization approach to eliminate redundant fragment shader executions on a mobile GPU. Different from these works, GRU focuses on GPGPU and efficiently realizing the partial GPU computing result reuse idea at a GPU kernel launch granularity.

7.6 Summary

In this chapter, we present GRU, a GPU sharing, result memoization and reuse ecosystem for high performance and cloud computing. GRU exploits computation and data redundancy seen in several important categories of GPU-accelerated workloads. We have fully implemented GRU and evaluation results show that GRU is effective in improving the turnaround time and reducing the GPU occupation time, while incurring a rather small runtime overhead.

CHAPTER 8

CONCLUSION

In this dissertation, we have presented five different works to improve the predictability of GPGPU computing in DNN-driven autonomous systems. Specifically, we propose GPES, a runtime system that allows GPU executions interruptible and preemptable in a multi-tasking environment. We proposed S^3DNN, a systemic solution that optimizes the execution of DNN workloads on GPU in a soft real-time multi-tasking environment. We proposed PredJoule, a runtime system which presents a layer-based approach that controls the timing and optimizes energy efficiency through exploiting each layer's performance/energy characteristics. In addition to the runtime systems we proposed, we investigate the problem of mapping multiple applications implemented using kernel graphs in a heterogeneous system, and present a theoretical framework that formulates this problem as an integer program and a set of practically efficient mapping algorithms. Furthermore we proposed a reuse-based approach to further improve the predictability of GPU computing.

REFERENCES

Abe, Y., H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres (2014, May). Power and performance characterization and modeling of gpu-accelerated systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 113–122.

Albericio, J., P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos (2016). Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, Volume 44, pp. 1–13. IEEE Press.

AMAZON (2006). Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2.

Arabnejad, H. and J. G. Barbosa (2014, March). List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Trans. Parallel Distrib. Syst. 25*(3), 682–694.

Arnau, J.-M., J.-M. Parcerisa, and P. Xekalakis (2014). Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In *ISCA'14*.

Auerbach, J., D. F. Bacon, P. Cheng, and R. Rabbah (2010). Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, New York, NY, USA, pp. 89–108. ACM.

Augonnet, C., S. Thibault, R. Namyst, and t. . S. b. . P. y. . . p. . . Wacrenier, Pierre-André.

Baek, W. and T. M. Chilimbi (2010). Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, Volume 45, pp. 198–209. ACM.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003). Xen and the art of virtualization. In *SOSP'03*.

Basaran, C. and K.-D. Kang (2012). Supporting preemptive task executions and memory copies in gpgpus. In *Proc. of IEEE ECRTS*, pp. 287–296.

Bhatotia, P., R. Rodrigues, and A. Verma (2012). Shredder: Gpu-accelerated incremental storage and computation. In *FAST'12*.

Bhatotia, P., A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin (2011). Incoop: Mapreduce for incremental computations. In *SOCC'11*.

Bhattacharya, S. and N. D. Lane (2016). Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pp. 176–189. ACM.

Bini, E., G. Buttazzo, and G. Lipari (2009). Minimizing cpu energy in real-time systems with discrete speed management. *ACM Transactions on Embedded Computing Systems (TECS) 8*(4), 31.

Bittencourt, L. F., R. Sakellariou, and E. R. Madeira (2010). Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pp. 27–34. IEEE.

Bojarski, M., D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.

Bu, Y., B. Howe, M. Balazinska, and M. D. Ernst (2010). Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow*.

Bueno, J., L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta (2011). Productive cluster programming with ompss. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, Berlin, Heidelberg, pp. 555–566. Springer-Verlag.

Canon, L.-C., E. Jeannot, R. Sakellariou, and W. Zheng (2008). Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pp. 73–84. Springer.

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09*.

Chen, C., A. Seff, A. Kornhauser, and J. Xiao (2015). Deepdriving: Learning affordance for direct perception in autonomous driving. In *ICCV*.

Chen, T. Y.-H., L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan (2015a). Glimpse: Continuous, real-time object recognition on mobile devices. In *SenSys*.

Chen, T. Y.-H., L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan (2015b). Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pp. 155–168. ACM.

Chen, W., J. Wilson, S. Tyree, K. Weinberger, and Y. Chen (2015). Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pp. 2285–2294.

Chen, Y.-H., J. Emer, and V. Sze (2016). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Volume 44, pp. 367–379.

Chi, P., S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie (2016). Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, Volume 44, pp. 27–39. IEEE Press.

Conners, D. and W.-M. Hwu (1999). Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO'99*.

Connors, D. A., H. C. Hunter, B.-C. Cheng, and W.-m. W. Hwu (2000). Hardware support for dynamic activation of compiler-directed computation reuse. In *ASPLOS'00*.

Currey, J., S. Baker, and C. Rossbach (2013). Supporting iteration in a heterogeneous dataflow engine.

Diamos, G. F. and S. Yalamanchili (2008). Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, New York, NY, USA, pp. 197–200. ACM.

Ding, Y. and Z. Li (2004). A compiler scheme for reusing intermediate computation results. In *CGO'04*.

Dowty, M. and J. Sugerman (2009). Gpu virtualization on vmware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev.*.

Duato, J., A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí (2010). rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *HPCS'10*.

Dudani, A., F. Mueller, and Y. Zhu (2002). Energy-conserving feedback edf scheduling for embedded systems with real-time constraints. In *ACM SIGPLAN Notices*, Volume 37, pp. 213–222. ACM.

Elliott, G. and J. H. Anderson (pp. 48-54, 2011). Real-world constraints of gpus in real-time systems. In *Proceedings of the First International Workshop on Cyber-Physical Systems, Networks, and Applications*.

Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman (2010, June). The pascal visual object classes (voc) challenge. *International Journal of Computer Vision 88*(2), 303–338.

Farrell, A. and H. Hoffmann (2016). Meantime: Achieving both minimal energy and timeliness with approximate computing. In *USENIX Annual Technical Conference*, pp. 421–435.

FREEDESKTOP (2012). Nouveau open-source driver. http://nouveau.freedesktop.org.

Fritsch, J., T. Kuehnl, and A. Geiger (2013). A new performance measure and evaluation benchmark for road detection algorithms. In *International Conference on Intelligent Transportation Systems (ITSC)*.

Fujii, Y., T. Azumi, N. Nishio, S. Kato, and M. Edahiro (2013). Data transfer matters for gpu computing. In *Proc. of IEEE International Conference on Parallel and Distributed Systems*, pp. 275–282.

Ghazal, A., T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen (2013). Bigbench: Towards an industry standard benchmark for big data analytics. In *SIGMOD'13*.

Girshick, R. (2015). Fast r-cnn. In *ICCV*.

Girshick, R., J. Donahue, T. Darrell, and J. Malik (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*.

Giunta, G., R. Montella, G. Agrillo, and G. Coviello (2010). A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par'10*.

Gottschlag, M., M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa (2013). Logv: Low-overhead gpgpu virtualization. In *FHC'13*.

Group, K. (2004). Opengl shading language. https://en.wikipedia.org/wiki/OpenGL_Shading_Language.

Group, K. O. W. (2008). Opencl-the open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl.

Gunda, P. K., L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang (2010). Nectar: Automatic management of data and computation in datacenters. In *OSDI'10*.

Han, S., X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally (2016). Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 243–254. IEEE.

Han, S., H. Mao, and W. J. Dally (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

Han, S., J. Pool, J. Tran, and W. Dally (2015). Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pp. 1135–1143.

Han, S., H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy (2016). Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*.

Harper, F. M. and J. A. Konstan (2015). The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*.

Harris, M. (2009). Gpgpu.org. http://gpgpu.org/aboute.

Heo, J., P. Jayachandran, I. Shin, D. Wang, T. Abdelzaher, and X. Liu (2011). Optituner: On performance composition and server farm energy minimization application. *IEEE Transactions on Parallel and Distributed Systems 22*(11), 1871–1878.

Hipp, D. R. (2018). Sqlite3 in-memory databases. https://www.sqlite.org/inmemorydb.html.

Hoffmann, H. (2014). Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pp. 223–232.

Hoffmann, H. (2015). Jouleguard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 198–214. ACM.

Hou, Y., J. Lai, and D. Mikushin (2011). Asfermi: An assembler for the nvidia fermi instruction set. http://code.google.com/p/asfermi.

Huang, K., L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo (2009). Adaptive dynamic power management for hard real-time systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pp. 23–32. IEEE.

Huang, S., J. Huang, J. Dai, T. Xie, and B. Huang (2010). The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW'10*.

Huang, S. S., A. Hormati, D. F. Bacon, and R. Rabbah (2008). Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, Berlin, Heidelberg, pp. 76–103. Springer-Verlag.

Hugo, A.-E., A. Guermouche, P.-A. Wacrenier, and R. Namyst (2013). Composing multiple starpu applications over heterogeneous machines: A supervised approach. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, Washington, DC, USA, pp. 1050–1059. IEEE Computer Society.

Huynh, L. N., Y. Lee, and R. K. Balan (2017). Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 82–95. ACM.

IBM (2016). Ibmsparkgpu. https://github.com/IBMSparkGPU/SparkGPU.

Imes, C., D. H. K. Kim, M. Maggio, and H. Hoffmann (2015, April). Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 75–86.

Jaderberg, M., A. Vedaldi, and A. Zisserman (2014). Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*.

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell (2014). Caffe: Convolutional architecture for fast feature embedding. In *ACM MM*.

Kang, Y., J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 615–629. ACM.

Kato, S. (2013). Implementing open-source cuda runtime. Technical report.

Kato, S., K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar (2011). Rgem: A responsive gpgpu execution model for runtime engines. In *Proc. of IEEE RTSS*, pp. 57–66.

Kato, S., K. Lakshmanan, R. Rajkumar, and Y. Ishikawa (2011). Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX ATC*.

Kato, S., M. McThrow, C. Maltzahn, and S. A. Brandt (2012). Gdev: First-class gpu resource management in the operating system. In *Proc. of USENIX Annual Technical Conference*, pp. 401–412.

Kenna, C. J., J. L. Herman, B. B. Brandenburg, A. F. Mills, and J. H. Anderson (2011). Soft real-time on multiprocessors: Are analysis-based schedulers really worth it? In *Proc. of IEEE RTSS*, pp. 93–103.

Kim, D. H. K., C. Imes, and H. Hoffmann (2015, Aug). Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, pp. 78–85.

Kim, Y.-D., E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin (2015). Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*.

Koren, Y., R. Bell, and C. Volinsky (2009). Matrix factorization techniques for recommender systems. *Computer*.

Koscielnicki, M. (2012). Envytools. git://0x04.net/envytools.git.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In *NIPS*.

Lab, P. D. (2015). Opencloud hadoop cluster trace. http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html.

Lagar-Cavilla, H. A., N. Tolia, M. Satyanarayanan, and E. De Lara (2007). Vmm-independent graphics acceleration. In *VEE*.

Lane, N. D., S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar (2016). Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*, pp. 1–12. IEEE.

Lattner, C. and V. Adve (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO'04*.

Lee, J., M. Samadi, Y. Park, and S. Mahlke (2013). Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proc. of IEEE PACT*, pp. 245–256.

LiKamWa, R., Y. Hou, J. Gao, M. Polansky, and L. Zhong (2016). Redeye: analog convnet image sensor architecture for continuous mobile vision. In *ACM SIGARCH Computer Architecture News*, Volume 44, pp. 255–266. IEEE Press.

Lin, T., M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, and j. . C. y. . . . v. . a. b. . d. b. . h. t. . W. u. . h. Piotr Dollár and C. Lawrence Zitnick, title = Microsoft COCO: Common Objects in Context.

Linetsky, M. (2001). *Programming Microsoft Directshow*. Wordware Publishing Inc.

Logothetis, D., C. Olston, B. Reed, K. C. Webb, and K. Yocum (2010). Stateful bulk processing for incremental analytics. In *SOCC'10*.

Luk, C., S. Hong, and H. Kim (2009). Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of ACM MICRO*, pp. 45–55.

Malka, M., N. Amit, M. Ben-Yehuda, and D. Tsafrir (2015). riommu: Efficient iommu for i/o devices that employ ring buffers. In *ASPLOS'15*.

McAuley, J., C. Targett, Q. Shi, and A. van den Hengel (2015). Image-based recommendations on styles and substitutes. In *SIGIR'15*.

McCool, M. D., A. D. Robison, and J. Reinders (2012). *Structured parallel programming: patterns for efficient computation*. Elsevier.

Michie, D. (1968). Memo functions and machine learning. *Nature*.

Mishra, N., H. Zhang, J. D. Lafferty, and H. Hoffmann (2015). A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *ACM SIGARCH Computer Architecture News*, Volume 43, pp. 267–281. ACM.

Mok, A. K.-L. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. Ph. D. thesis, Massachusetts Institute of Technology.

Murray, D. G., M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand (2011). Ciel: A universal execution engine for distributed data-flow computing. In *NSDI'11*.

National institute for research in computer science and control (2008). *How To Optimize Performance With StarPU*.

NVIDIA (2003). C for graphics. https://en.wikipedia.org/wiki/Cg.

NVIDIA (2010). Nvidia gf100 whitepaper. http://www.nvidia.com/object/IO_86775.html.

NVIDIA (2011). Cuda 4.0. http://developer.nvidia.com/cuda-toolkit-40.

NVIDIA (2014). Nvidia kepler architecture. http://www.nvidia.com/object/nvidia-kepler.html.

NVIDIA (2015). Cuda 7 streams simplify concurrency. https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

NVIDIA (2016). Drive px 2. http://www.nvidia.com/object/drive-px.html.

OpenACC (2013). The openacc application programming interface. https://www.openacc.org/.

Pai, S., M. J. Thazhuthaveetil, and R. Govindarajan (2013). Improving gpgpu concurrency with elastic kernels. In *Proc. of ACM SIGPLAN*, pp. 407–418. ACM.

Peng, D. and F. Dabek (2010). Large-scale incremental processing using distributed transactions and notifications. In *OSDI'10*.

Popa, L., M. Budiu, Y. Yu, and M. Isard (2009). Dryadinc: Reusing work in large-scale computations. In *HotCloud'09*.

Pugh, W. and T. Teitelbaum (1989). Incremental computation via function caching. In *POPL'89*.

Reagen, B., P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks (2016). Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, Volume 44, pp. 267–278. IEEE Press.

Redmon, J., S. Divvala, R. Girshick, and A. Farhadi (2016). You only look once: Unified, real-time object detection. In *CVPR*.

Ren, S., K. He, R. Girshick, and J. Sun (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS*.

Romero, A., N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio (2014). Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*.

Rossbach, C. J., J. Currey, M. Silberstein, B. Ray, and E. Witchel (2011). Ptask: Operating system abstractions to manage gpus as compute devices. In *Proc. of ACM SOSP*, pp. 233–248.

Rossbach, C. J., Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly (2013). Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP'13*.

Sakellariou, R. and H. Zhao (2004). A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pp. 111. IEEE.

Samadi, M., D. A. Jamshidi, J. Lee, and S. Mahlke (2014). Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS'14*.

Santriaji, M. H. and H. Hoffmann (2016). Grape: Minimizing energy for gpu applications with performance requirements. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13. IEEE.

Shafiee, A., A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar (2016). Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News 44*(3), 14–26.

Shi, L., H. Chen, J. Sun, and K. Li (2012). vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*.

Simonyan, K. and A. Zisserman (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Sodani, A. and G. S. Sohi (1997). Dynamic instruction reuse. In *ISCA'97*.

Sorber, J., A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger (2007). Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pp. 161–174. ACM.

STP (2014). The simple theorem prover. http://stp.github.io/.

Suzuki, Y., S. Kato, H. Yamada, and K. Kono (2014). Gpuvm: Why not virtualizing gpus at the hypervisor. In *ATC'14*.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Tesla (2017). Tesla: Challenges in fleet learning. https://seekingalpha.com/article/4045423-tesla-challenges-fleet-learning.

Thies, W., M. Karczmarek, and S. P. Amarasinghe (2002). Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, London, UK, UK, pp. 179–196. Springer-Verlag.

Tian, K., Y. Dong, and D. Cowperthwaite (2014). A full gpu virtualization solution with mediated pass-through. In *ATC'14*.

Topcuouglu, H., S. Hariri, and M.-y. Wu (2002, March). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst. 13*(3), 260–274.

Umuroglu, Y., N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers (2017). Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74. ACM.

Verner, U., A. Schuster, and M. Silberstein (2011). Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the international conference on Supercomputing*, pp. 120–129. ACM.

Wang, K., X. Ding, R. Lee, S. Kato, and X. Zhang (2014). Gdm: Device memory management for gpgpu computing. In *Proc. of ACM SIGMETRICS*, pp. 533–545.

Wang, L., J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu (2014). Bigdatabench: A big data benchmark suite from internet services. In *HPCA'14*.

Weinsberg, Y., D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff (2008). Tapping into the fountain of cpus: on operating system support for programmable devices. *ACM SIGOPS Operating Systems Review 42*(2), 179–188.

Wu, J., A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt (2016). gpucc: an open-source gpgpu compiler. In *CGO'16*.

Xu, M., F. Qian, and S. Pushp (2017). Enabling cooperative inference of deep learning on wearables and smartphones. *arXiv preprint arXiv:1712.03073*.

Xue, J., J. Li, D. Yu, M. Seltzer, and Y. Gong (2014). Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 6359–6363. IEEE.

Yang, T.-J., Y.-H. Chen, and V. Sze (2017). Designing energy-efficient convolutional neural networks using energy-aware pruning. *arXiv preprint*.

Y.C. (2012). Extremely fast non-cryptographic hash algorithm. https://code.google.com/archive/p/xxhash.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica (2010). Spark: cluster computing with working sets. In *HotCloud'10*.

Zhang, H. and H. Hoffmann (2016). Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, New York, NY, USA, pp. 545–559. ACM.

Zhao, H. and R. Sakellariou (2003). An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Euro-Par*, pp. 189–194.

Zhao, H. and R. Sakellariou (2006). Scheduling multiple dags onto heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 14–pp. IEEE.

Zhou, H., Y. Fu, and C. Liu (2015). Supporting dynamic gpu computing result reuse in the cloud. In *HotCloud*.

Zhou, H., G. Tong, and C. Liu (2015). Gpes: a preemptive execution system for gpgpu computing. In *RTAS*.

BIOGRAPHICAL SKETCH

Husheng Zhou received his BS degree in Software Engineering from China University of Petroleum in 2007, his MS degree in Software Engineering from Peking University in 2011, and his PhD degree in Computer Science from The University of Texas at Dallas in 2018. His research interests include GPGPU, real-time and embedded systems, heterogeneous systems, and high performance computing.

CURRICULUM VITAE

Husheng Zhou

October 2018

Educational History:

B.S., Computer Science, China University of Petroleum, 2007
M.S., Software Engineering, Peking University, 2011
Ph.D., Computer Science, University of Texas at Dallas, 2018

Predictable GPGPU Computing in DNN-driven Autonomous Systems
Ph.D. Dissertation
Computer Science Department, University of Texas at Dallas
Advisors: Dr.Cong Liu

Working Experience:

Summer Research Intern, IBM T. J. Watson Research Center, May 2016 – August 2016
Summer Intern, Huawei Technologies Co., Ltd, May 2014 – August 2014
Research Engineer, Singapore Management University, August 2011 – August 2012

Professional Activities:

SIGBED 2018 Best Paper Award, CPS Week 2018, 2018
SIGBED CPSWEEK 2015 Student Travel Grant, CPS Week 2015, 2015
Usenix 2015 Student Travel Grant, Usenix ATC 2015, 2015
ACM CCS 2017 Volunteer, ACM CCS 2017, 2017

Publications:

1. Husheng Zhou*, Soroush Bateni*, Cong Liu. PredJoule: A Timing-Predictable Energy Optimization Framework for Deep Neural Networks. IEEE Real-Time Systems Symposium (RTSS), 2018.

2. Husheng Zhou, Soroush Bateni, Cong Liu. Exploring Computation and Data Redundancy via Partial GPU Computing Result Reuse. ACM International Conference on Supercomputing (ICS), 2018.

3. Husheng Zhou, Soroush Bateni, Cong Liu. S^3DNN: Supervised Streaming and Scheduling for GPU-accelerated Real-Time DNN Workloads. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2018. **[Best Paper Award]**

4. Zheng Dong, Yuchuan Liu, Husheng Zhou, Xusheng Xiao, Yu Gu, Lingming Zhang, Cong Liu. An Energy-efficient Offloading Framework with Predictable Temporal Correctness. IEEE Symposium on Edge Computing (SEC), 2017.

5. Husheng Zhou, Yangchun Fu and Cong Liu. Supporting Dynamic GPU Computing Result Reuse in the Cloud. USENIX Conference on Hot Topics in Cloud Computing (HotCloud), 2015.

6. Husheng Zhou, Guangmo Tong and Cong Liu. GPES: A Preemptive Execution System for GPGPU Computing. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015.

7. Wen-Hung Huang, Husheng Zhou, Jian-Jia Chen and Cong Liu. PASS: Priority Assignment of Real-Time Tasks with Dynamic Suspending Behavior under Fixed-Priority Scheduling. Design Automation Conference (DAC), 2015.

8. Husheng Zhou and Cong Liu. Task Mapping in Heterogeneous Embedded Systems for Fast Completion Time. ACM International Conference on Embedded Software (EMSOFT), 2014.

Technique Skills:

Languages: C/C++, CUDA, Java, Linux Shell, C#, Ocaml, Scala, Python
Tools: Spark, Caffe, YOLO, Tensorflow, Xen, Qemu, GPGPU-sim, LLVM, IDA Pro, OllyDbg, Valgrind, Nouveau, MySQL, Subversion, Git, Z3 SMT Solver
Hardware and Systems: Linux, ROS, Litmus RT , Windows, Mac, NVIDIA Jetson TX2, Raspberry Pi