THE UNIVERSITY OF CHICAGO

EXPLOITING THREAD SLACK TIME FOR ENERGY SAVING IN GPU’S APPLICATIONS
WITH HARD REAL TIME REQUIREMENTS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER’S

DEPARTMENT OF COMPUTER SCIENCE

BY
MUHAMMAD H. SANTRIAJI

CHICAGO, ILLINOIS
2019
To The Almighty God, Tika, and my family.
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. vi
LIST OF TABLES ............................................................... vii
ACKNOWLEDGMENTS ........................................................... viii
ABSTRACT ................................................................................ ix

1 INTRODUCTION .................................................................... 1

2 BACKGROUND AND MOTIVATION ........................................... 5
   2.0.1 GPU Scheduling Overview .............................................. 5
   2.0.2 Opportunities for Energy Savings ..................................... 6
   2.0.3 Challenges for Fine-grain Slack Transfer ......................... 9

3 METHODOLOGY .................................................................... 11
   3.1 Baseline Architecture ...................................................... 11
   3.2 Benchmarks .................................................................... 12
   3.3 Points of Comparison ...................................................... 12

4 MERLOT DESIGN AND IMPLEMENTATION ............................... 13
   4.1 Software-Hardware Interface ............................................. 14
   4.2 Checkpoint Handler ........................................................ 15
   4.3 Optimizer ....................................................................... 16
   4.4 Resource Allocator .......................................................... 19
   4.5 Using MERLOT for Dynamic Timing Analysis ..................... 19
   4.6 Hardware Implementation ............................................... 20
   4.7 Real-Time Considerations ............................................... 21

5 RESULTS .............................................................................. 22
   5.1 Meeting Hard Deadlines .................................................. 22
      5.1.1 Performance ............................................................. 22
      5.1.2 Energy Saving ........................................................... 24
   5.2 Comparison of Hard and Soft Deadline .............................. 24
   5.3 Sensitivity Analysis .......................................................... 26

6 RELATED WORK ................................................................... 29

7 CONCLUSION AND FUTURE WORK ...................................... 32
   7.1 Conclusion ..................................................................... 32
   7.2 Future Works ............................................................... 32
LIST OF FIGURES

2.1 A GPU application with two kernels (a), each of which is divided into cooperative thread arrays (CTAs). The CTAs are scheduled in hardware (b), which assigns CTAs to SMs. .......................................................... 5

2.2 Slack transfer for energy savings in the GPU application from Figure 2.1(a). The application has a deadline. When kernel 1 finishes ahead of its worst case timing, the resulting slack can be transferred to kernel 2, which can now use a reduced energy resource allocation without risking a timing violation, even if it encounters a worst case input. .......................................................... 7

2.3 backprop’s performance (a) (in CTAs completed per millisecond), (b) resource usage and power (in Watts) (c) for three different scheduling algorithms. Each scheduler meets the timing requirement of completing the application in 1 ms. Energy is the area under each curve in (b). The software slack scheduler reduces energy by just 1% compared to race-to-idle, but MERLOT achieves a 15.4% reduction. ............. 9

4.1 MERLOT block diagram .................................................. 14

5.1 Running time normalized to deadline. The left figure shows the results where the worst case is \(1.5 \times\) the average case and the right side shows the case where the ratio is \(2.0 \times\). 23

5.2 Energy normalized to the race-to-idle approach. The left figure shows the results where the worst case is \(1.5 \times\) the average case and the right side shows the case where the ratio is \(2.0 \times\). .................................................. 23

5.3 MERLOT and GRAPE Performance .................................. 25

5.4 MERLOT and GRAPE Energy ........................................... 26

5.5 MERLOT in 5X, 10X and 15X WCET ............................... 27
# LIST OF TABLES

<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>GPGPU-Sim Configuration.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Benchmarks Used.</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Notation used in the thesis.</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Resource Allocator Translation</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>Average Deadline vs Energy Savings.</td>
<td>27</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would first like to thank my thesis advisor Dr. Henry Hoffmann at University of Chicago. The door to Prof. Hoffmann office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this theses to be my own work, but steered me in the right the direction whenever he thought I needed it.
ABSTRACT

Correct functioning of embedded systems requires strict timing guarantees. Traditionally, enforcing timing guarantees is the operating system’s responsibility. The OS scheduler assigns sufficient resources to an application task to ensure it meets its deadline. Meeting hard real-time deadlines requires the scheduler to be conservative and allocate for the worst case timing; when behavior is not worst case, extra resources are allocated and energy is wasted. Some software schedulers reduce this energy waste by recognizing when an application is ahead of a worst case schedule and reclaiming unneeded resources, but they are fundamentally limited by (1) overhead and (2) a lack of visibility into low-level resource usage. Therefore, this thesis advocates hardware assistance for energy management of hard real-time tasks. Specifically, we propose MERLOT, a hardware-based resource manager for GPUs that enforces software-specified timing guarantees with minimal energy. We implement MERLOT in VHDL and find that its performance, power, and area overheads are minuscule. We implement MERLOT in GPGPU-Sim to test timing and energy consumption and compare to two software-only approaches: one that always allocates for worst case timing and an intelligent approach that reduces resource usage when it recognizes better than worst case behavior. Compared to the approach that always allocates for worst case, MERLOT reduces energy by 16.73% on average, with 16.43% in 1.5X and 17.03% in 2.0X WCET. Compared to the intelligent software-only approach, MERLOT produces a 15.6% energy savings. MERLOT uses less energy than software-only approaches because it recognizes better than worst case behavior earlier, by monitoring hardware events that are not visible to software and quickly adjusting resource usage.
CHAPTER 1
INTRODUCTION

Systems such as self-driving cars have enormous computational requirements and incorporate GPUs ([18, 65]) and emerging GPU-like accelerators (e.g., for deep neural network sensor processing [50]) to meet those computational demands. A number of software schedulers have arisen for managing these GPU and accelerator resources to ensure complex embedded applications—like autonomous vehicles—meet the timing requirements necessary to ensure correct operation [18, 32, 31, 17, 44]. A secondary priority for these systems is minimizing energy consumption given the timing constraints.

There is a natural tension between timing guarantees and energy efficiency. When timing guarantees cannot be violated—i.e., hard real-time requirements—software must conservatively allocate resources for worst case [9]. This conservatism is wasteful when inputs do not exercise the worst case execution path, consuming more energy than necessary [34].

Several software approaches have arisen to augment real-time schedulers with energy reduction [14, 60, 26, 7, 4, 21, 67, 30]. While these schedulers all differ in their details, the unifying theme is the exploitation of timing slack. If the scheduler can detect a task will terminate before its worst case schedule, the resulting slack can be converted into energy savings. In a GPU application consisting of multiple kernels, if an early kernel finishes ahead of schedule, then the slack can be transferred to the next kernel, slowing it down to save energy without risking the overall application timing.

Implementing slack transfer requires the application to signal sub-task completion to the software scheduler. This need for signaling, or checkpointing, limits the energy software can save through slack transfer. While kernels form a natural checkpoint, GPU applications may have only a few kernels (or even just one). Each kernel, however, consists of thousands of threads, making it tempting to use thread termination as a checkpoint. Signaling thread completion to a software scheduler, however, requires making significant changes to GPU code to trigger fine-grain commu-
nication between the GPU and CPU. Unfortunately, the overhead of this increased communication will quickly overwhelm any energy savings.

Consequently, this thesis advocates hardware support for managing GPU resources to meet hard real-time deadlines with minimal energy by observing and managing timing slack at a sub-kernel level. This hardware support is not designed to replace software scheduling, but to complement it by extracting extra energy savings not available to software schedulers. There are two advantages to hardware-based resource management for real-time processing and one challenge to be addressed. The potential advantages are:

1. **Fast reaction time.** Hardware can perform fine-grained, low-overhead checkpointing to determine definitively when a kernel is not worst case and reduce its resource usage. Software cannot effectively detect or adapt to timing slack below the kernel level, because it cannot know a kernel is ahead of schedule until it completes.

2. **Knowledge of resource usage.** Hardware has the most up-to-date information on a kernel’s resource usage. Even a worst case kernel rarely requires all resources. One kernel may require more memory bandwidth, while a different one requires more compute. By observing usage, hardware can tailor resources to minimize energy while ensuring timing requirements.

The challenge of hardware-based resource management for real-time systems, however, is that software has all the information about timing requirements and progress. For example, it is software that knows task periods and deadlines. Hardware resource management is also not appropriate for all platforms, such as time-shared general-purpose processors, where tasks are not required to be decomposed into identical threads operating on different data. A hardware approach does make sense, though, for the growing class of specialized accelerators that have been proposed to increase performance and energy efficiency at the end of Dennard scaling [64, 19].

To enable hardware resource management in GPUs, we propose a small change in instruction set architecture that informs hardware of timing constraints. We show how to use this information to build a hardware resource allocator that provides hard timing guarantees with lower energy than
state-of-the-art software-only approaches. While we think this approach is generally applicable to a number of accelerators—including recently released GPU-like accelerators for deep neural network based sensor processing [50]—we implement and evaluate it for GPUs.

We call our system MERLOT. It consists of an interface allowing software to communicate timing requirements to hardware and a hardware resource manager that automatically adjusts (1) GPU frequency, (2) memory frequency, and (3) GPU core usage. The interface is a set of registers software uses to specify a GPU kernel’s deadline and worst case execution time for a number of checkpoints hardware should take within the kernel. MERLOT begins executing the kernel with all resources available. It then samples execution—measuring progress as completed thread blocks—at fixed time intervals to determine (1) whether the current kernel is worst case and (2) which resources the kernel actually needs to meet its deadline. MERLOT then reduces the resources in use so that the software-specified deadline is met while energy consumption is minimized.

We implement MERLOT in VHDL and synthesize for an FPGA to show it can be implemented in hardware. To demonstrate it meets hard real-time requirements on a GPU, we integrate the design into GPGPU-Sim, a cycle-accurate simulator of an NVIDIA GTX 480 GPU [5]. We compare the timing guarantees and energy consumption to prior software-only approaches that either (1) race-to-idle—completing the kernel as fast as possible and then idling the GPU—or (2) transfer timing slack between kernels [7, 4, 21].

We test MERLOT with a wide range of applications and latency targets. MERLOT meets hard real-time deadlines. Compared to race-to-idle, MERLOT reduces energy consumption by 16.43%. Compared to the sophisticated software scheduler that transfers slack between kernels, MERLOT reduces energy consumption by 15.63%. Compared to a prior hardware approach that provides only soft timing guarantees, MERLOT provides almost equivalent energy savings without missing deadlines. MERLOT provides energy savings even when deadlines are large multiples of the worst case latency. Finally, experiments with modified power models show that MERLOT’s energy savings will increase as GPUs incorporate more energy efficient features and lower-power
idle states.

MERLOT’s primary contribution is recognizing the potential for energy savings by incorporating hardware support into hard real-time scheduling. This energy savings arises from hardware’s ability to quickly detect (1) when inputs are not worst case and (2) what resources are actually needed by the current task. Once 1 and 2 are known, hardware can scale back resource usage without violating the timing constraints. We demonstrate this benefit for GPUs, but we believe the idea is widely applicable to the plethora of accelerators that have recently been proposed for various specialized tasks. We release our modifications to GPGPU-Sim as open source so that others can recreate or extend our results.\footnote{https://github.com/santriaji/MERLOT.git}
CHAPTER 2
BACKGROUND AND MOTIVATION

We briefly review CPU-GPU interaction with a focus on what units are schedulable in software versus hardware. We then review how timing slack in GPU applications can be turned into energy savings with intelligent software scheduling. With this background in place, we argue that even greater energy saving is possible with a hardware assist to perform this slack transfer at an even finer granularity—using hardware schedulable units instead of software schedulable units.

2.0.1 GPU Scheduling Overview

Figure 2.1(a) presents a high-level overview of a GPU program. Programmers create software applications that run on a traditional CPU and offload significant computation to the GPU, which provides a much faster and more energy efficient platform for highly parallel, regularly structured computations. The pieces of computation that are offloaded are called kernels, which correspond to parallel loops in traditional programs. Kernels themselves are broken up into cooperative thread arrays (CTAs), or blocks of threads that are executed in single-instruction multiple-data style on the GPU’s streaming multiprocessors (SMs), analogous to CPU cores.

The kernel is the lowest-level software schedulable unit in a GPU program. Software running on a CPU (whether real-time or not) is responsible for launching the kernel on the GPU and syn-

![Figure 2.1: A GPU application with two kernels (a), each of which is divided into cooperative thread arrays (CTAs). The CTAs are scheduled in hardware (b), which assigns CTAs to SMs.](image)

5
chronizing to ensure the kernel completes before the results are read from shared memory. Many software schedulers exist that allow GPUs to be shared among multiple processes [55, 33], and real-time schedulers exist that manage applications and kernels to ensure predictable timing in GPU-augmented systems [32, 31, 17, 18].

Note, however, that while software specifies CTAs, it has no control over—or even visibility into—how they are scheduled on the GPU hardware. While a GPU may have dozens of SMs, a typical kernel will have 100s to 1000s of CTAs. Therefore, GPU hardware maintains its own scheduling queue (as shown in Figure 2.1(b)). As CTAs complete, hardware is responsible for selecting the next CTA to run. While hardware-level schedulers have been proposed [61], we know of only one that can provide soft timing guarantees [57], and we are not aware of any that support hard timing requirements.

Thus, in GPU-based systems there is divided knowledge. Software has information about application structure and kernel timing requirements that hardware does not. Hardware, however, can observe kernel progress at the much finer granularity of CTA completion. Additionally, hardware has knowledge about what resources are actually in use; e.g., whether the kernel compute- or memory-bound. The primary goal of this thesis is to bridge this knowledge gap so that software can provide timing information to hardware, and hardware can use that information to reduce energy while ensuring software-specified timing requirements are met.

2.0.2 Opportunities for Energy Savings

Schedulers providing hard timing guarantees are necessarily conservative [9]. The most conservative approach is to exploit dynamic power management (DPM) by always allocating all resources required to meet timing constraints and transitioning the system to a low-power idle state if a task is not worst case [14, 60]. Prior work, however, shows that this race-to-idle strategy is never better than a more intelligent strategy that knows what resources are required for the current case [34]. The problem, of course, is determining that the current case can be slowed down without violating
Slack Transfer

A number of schedulers have been developed that save energy in hard real-time systems by exploiting \textit{slack} [67, 30, 7, 4, 21]. If a task can be decomposed into parts and the scheduler can recognize that one part has finished ahead of schedule (\textit{i.e.,} is not worst case) then this extra time—\textit{i.e.}, slack—can be transferred to the next part of the task, allowing the scheduler to slow that part down without risking violation of the overall task’s timing requirements.

Figure 2.2 illustrates this concept of \textit{slack transfer} as applied to the example GPU application from Figure 2.1(a). In this case, the application has a hard timing deadline. A software scheduler can monitor the execution of kernel 1 to determine that it finished ahead of its worst case timing. This slack is transferred to kernel 2, as the scheduler allocates fewer resources—slowing down kernel 2—without violating the deadline on the application itself. If kernel 2 is also not worst case, then the system can idle the GPU until the next application task is ready to be scheduled.
Fine-grain Slack Transfer

Figure 2.2 highlights the energy savings possible with slack transfer, but it also hints that more energy savings may be available by incorporating hardware-level knowledge into the transfer process. Consider that if the application consisted of only a single kernel, there would be no opportunity to transfer slack. Furthermore, even in the multi-kernel scenario, our intuition is: if kernel 1 is not worst case, then that property is probably observable early in the execution of kernel 1’s CTAs, so additional energy could have been saved by reducing resource usage as soon as CTAs complete ahead of the worst case schedule. Given that a single kernel can consist of 100s–1000s of CTAs, there is significantly more opportunity to transfer slack at this level.

We test this intuition empirically using the backprop application (an important part of deep neural network computation [13]). backprop consists of two kernels. To illustrate the potential energy savings we run it on an instrumented GPU simulator to measure its performance and power as a function of time (see section 3 for details on our experimental setup). We compare three scheduling algorithms that guarantee timing while reducing energy. The first, race-to-idle, simply allocates all GPU resources to the application and then idles the GPU until the deadline (when, presumably, the next task will be scheduled). The second software approach transfers slack from the first kernel to the second, reducing energy usage. The third is the proposed contribution of this thesis, MERLOT, which transfers slack at the CTA-level in hardware.

Figure 2.3(a) shows the resource usage which proportional to performance over time, while Figure 2.3(c) shows the power over time. First, the two different kernels are easily visible in the time series data race-to-idle spends maximum resource in the first and second kernel. Additionally, we see that all three approaches meet the 1 ms deadline. Considering power consumption, race-to-idle has the highest peak power, which it maintains for the duration of the second kernel. The software slack scheduler recognizes the first kernel completes ahead of schedule and reduces power for the second kernel—by reducing its resource usage from what would be required if the second kernel started at the first kernel’s worst case completion time. MERLOT’s hardware-based
Figure 2.3: backprop’s performance (a) (in CTAs completed per millisecond), (b) resource usage and power (in Watts) (c) for three different scheduling algorithms. Each scheduler meets the timing requirement of completing the application in 1 ms. Energy is the area under each curve in (b). The software slack scheduler reduces energy by just 1% compared to race-to-idle, but MERLOT achieves a 15.4% reduction.

The above results demonstrate the energy saving potential of hardware-based slack transfer on GPUs. Achieving these benefits, however, requires addressing two challenges: overhead and resource allocation.

Measuring application progress at the CTA level necessarily requires more fine grain monitoring. GPU software interfaces do not currently support observing the status of a single CTA. It might be possible, however, for the programmer to augment GPU code such that each thread signals completion to the CPU through an interrupt or a write to shared memory. The problem is that the huge number of threads in the kernel would quickly overwhelm software, if each independently signaled termination. We therefore argue that sub-kernel slack transfer requires hardware
support. Adding this support is reasonable because hardware GPU schedulers already monitor CTA progress (see Figure 2.1(b)). Of course, the additional hardware for resource management must be low-overhead in terms of its area, performance, and power.

Resource allocation is a challenge because hardware must be able to tell with certainty how to adjust a kernel's resource usage without violating timing constraints. For this challenge we assume it is beyond software's scope to provide hints about what resources the application might use. It is up to hardware to determine an appropriate resource allocation at runtime.
CHAPTER 3
METHODOLOGY

3.1 Baseline Architecture

We evaluate timing using GPGPU-Sim v3.2.2, a cycle-accurate GPU Simulator [5] and energy using GPUWattch [38], which models GPU power and energy consumption. This tool set has been proven to deliver realistic timing and energy that is within a few percent of the timing on real hardware. Modifying such a cycle accurate simulator is the closest we can get to evaluating MERLOT’s hardware design without implementing the hardware itself. Our baseline implementation in GPGPU-Sim uses the configuration in Table 3.1.

Table 3.1: GPGPU-Sim Configuration.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader Core</td>
<td>Fermi 15 Shader Cores, 5-Stage Pipeline,</td>
</tr>
<tr>
<td>Resources / Core</td>
<td>Max 1536 Threads, 32 kB Shared Memory, 32768 Registers, 48 warps</td>
</tr>
<tr>
<td>L1 Caches / Core</td>
<td>32kB 8-way L1 Data Cache, 8 kB 4-way texture, 8kB 4-way constant cache, 64B Line Size</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>786 kB, 128B Line size, 8-way associative</td>
</tr>
<tr>
<td>Scheduling</td>
<td>GTO Scheduling, Load Balance scheduling</td>
</tr>
<tr>
<td>Interconnect</td>
<td>1 crossbar, 16B Channel Width,</td>
</tr>
<tr>
<td>DRAM Model</td>
<td>FR-FCFS (128 RQS/MC), 4B Bus Width, 4 DRAM-banks/MC, 2 kB page size, 4 burst size, 8 MCs</td>
</tr>
</tbody>
</table>

In this implementation, we have a set of SM actuators, SM frequency actuators and memory frequency actuators. SM actuators have 16 members covering operation from idle to 15 SMs. We assume that SM frequency actuator has 7 P-states, ranging from a peak of 700 MHz to a minimum of 100 MHz, with step size of 100 MHz. These settings align with the GTX 480’s existing DVFS
Table 3.2: Benchmarks Used.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Abbr.</th>
<th>Resource Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpropagation [12]</td>
<td>BP</td>
<td>Memory</td>
</tr>
<tr>
<td>Fast Fourier Transform [63]</td>
<td>FFT</td>
<td>Memory</td>
</tr>
<tr>
<td>Needelman-Wunch [12]</td>
<td>NW</td>
<td>Unsaturated</td>
</tr>
<tr>
<td>Breadth First Search [12]</td>
<td>BFS</td>
<td>Cache</td>
</tr>
<tr>
<td>Needleman-Wunch [12]</td>
<td>NW</td>
<td>Unsaturated</td>
</tr>
<tr>
<td>Breadth First Search [12]</td>
<td>BFS</td>
<td>Cache</td>
</tr>
<tr>
<td>Gaussian Elimination [12]</td>
<td>GE</td>
<td>Unsaturated</td>
</tr>
<tr>
<td>Hotspot [12]</td>
<td>HOT</td>
<td>Compute</td>
</tr>
<tr>
<td>LU Decomposition [12]</td>
<td>LUD</td>
<td>Unsaturated</td>
</tr>
<tr>
<td>Particle Filter Float [12]</td>
<td>PFF</td>
<td>Memory</td>
</tr>
<tr>
<td>Pathfinder [12]</td>
<td>PF</td>
<td>Compute</td>
</tr>
<tr>
<td>Sum of Absolute Diff. [63]</td>
<td>SAD</td>
<td>Unsaturated</td>
</tr>
<tr>
<td>Distance-Cutoff Coulombic Potential [63]</td>
<td>CUTCP</td>
<td>Compute</td>
</tr>
</tbody>
</table>

3.2 Benchmarks

We evaluate the benchmarks in Table 3.2. For each, we collect timing from the simulator and power from GPUWattch.

3.3 Points of Comparison

We compare MERLOT to two different scenarios: (1) the race-to-idle heuristic which runs each task in the fastest configuration available and then puts the system into a low-power idle state until the next job is to be processed [34] and (2) a software approach which uses a combination of slack transfer, DVFS and low-power idle states to adjust the configuration between the kernels. This second approach is based on implementing existing software approaches (e.g., [67, 30, 7, 4, 21]) for the GPU. We set the idle power to 23.31 W, which is the idle power when the frequency and voltage of both GPU and DRAM is set as low as possible.
CHAPTER 4
MERLOT DESIGN AND IMPLEMENTATION

MERLOT provides architectural support for meeting hard real-time deadlines while minimizing energy through management of a GPU’s (1) streaming multiprocessors (SMs or cores), (2) SM frequency, and (3) DRAM frequency. We emphasize that MERLOT’s goal is not to replace existing software schedulers, but to augment them by ensuring the software-specified timing requirements are met while adjusting the specified resources to reduce energy. MERLOT operates on hardware-level structures that are simply not accessible to software. Thus, MERLOT allows existing software schedulers to focus on high-level scheduling decisions (e.g., when to schedule a kernel), while hardware focuses on low-level resource management, for which it is better suited.

Figure 4.1 illustrates MERLOT’s design. When software launches a kernel on the GPU, it specifies a number of checkpoints (in units of CTAs) and a worst case completion time (in milliseconds) for each. These values are written into hardware registers on the GPU. As the kernel executes, MERLOT measures the time at each checkpoint to determine whether or not the kernel is ahead of its worst case timing. If the kernel is ahead, then this timing slack can be transferred to the next checkpoint; i.e., MERLOT adjusts hardware resources to slow the next set of CTAs down such that energy efficiency is maximized and the deadline is just met. MERLOT’s hardware consists of three modules: (1) a checkpoint handler that tracks execution progress and determines available slack; (2) an optimizer that turns the slack into specific settings for the number of SMs to use, the SM frequency, and the DRAM frequency; and (3) a resource allocator that actually enforces the settings determined by the optimizer.

We detail each MERLOT module in turn. For each module we give an intuitive overview and then formally specify its behavior in the form of equations and algorithms. The final subsections discuss how MERLOT’s hardware structures can be trivially repurposed for timing analysis instead of enforcement and then describe MERLOT timing guarantees and implementation issues. Table 4.1 summarizes the notation used throughout this section.
Table 4.1: Notation used in the thesis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nCTA_j$</td>
<td>number of CTAs in checkpoint $j$</td>
</tr>
<tr>
<td>$wCTA_j$</td>
<td>WCET of checkpoint $j$</td>
</tr>
<tr>
<td>$t_{current}$</td>
<td>current time</td>
</tr>
<tr>
<td>$o$</td>
<td>timing overhead</td>
</tr>
<tr>
<td>$t_{ahead}$</td>
<td>timing slack</td>
</tr>
<tr>
<td>$\eta$</td>
<td>slowdown factor ranged from 0 to 1</td>
</tr>
<tr>
<td>$f_{max}$</td>
<td>maximum frequency id</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>frequency id ranged from 0 to $f_{max}$</td>
</tr>
<tr>
<td>$f_{crit}$</td>
<td>critical frequency for energy efficiency</td>
</tr>
<tr>
<td>$ALU_{util}$</td>
<td>ALU utilization in GPU, ranged from 0 to 1</td>
</tr>
<tr>
<td>$th_{ALU}$</td>
<td>a threshold to distinguish memory and compute needs</td>
</tr>
<tr>
<td>$sm$</td>
<td>status of SM in GPU</td>
</tr>
<tr>
<td>$nSM$</td>
<td>number of SMs</td>
</tr>
<tr>
<td>$f_{mem}$</td>
<td>memory frequency of configuration</td>
</tr>
<tr>
<td>$f_{SM}$</td>
<td>SM frequency of configuration</td>
</tr>
</tbody>
</table>

Figure 4.1: MERLOT block diagram

4.1 Software-Hardware Interface

As mentioned in section 2, an exclusively software approach can only transfer slack between kernels. MERLOT allows further energy savings by tracking CTA completions in hardware as checkpoints within a kernel, allowing slack to be transferred within a kernel at a finer granularity than software alone can handle. To ensure timing, however, hardware must be informed of software’s requirements.

The interface between software and hardware is a set of special purpose registers on the GPU that allow software to specify the number of checkpoints (in terms of number of CTAs to complete) and the deadline for each checkpoint (in terms of the worst case timing for that set of CTAs). A checkpoint $j$ is a pair: the number of CTAs in a checkpoint $nCTA_j$ and the worst case timing for
that set of CTAs $wCTA_j$. To keep the memory overhead low, we bound the number of checkpoints to 16 per application in this implementation.

We believe this is a flexible interface that supports a wide range of software schedulers. Using this simple interface, software can specify checkpoints per application or per kernel by simply dividing existing timing requirements among the checkpoints. More sophisticated schedulers with deep knowledge of the underlying hardware and data access patterns can even split timing requirements unevenly among CTAs. A methodology for worst case timing analysis of kernels, CTAs, and threads in GPU applications can be found in [6]. Notably, as MERLOT tracks individual CTAs’ completion time, MERLOT can be trivially extended to produce per CTA timing reports (as described in Section III.E). Using this profiling mode, a developer could first use MERLOT to better understand CTA timing and then switch to enforcement mode using the deadlines produced from profiling.

### 4.2 Checkpoint Handler

Algorithm 1 lines 3–9 show pseudo-code for the checkpoint handler. Starting from the last checkpoint, MERLOT counts CTA completion in hardware. When the count is equal to a checkpoint count (i.e., when the required number of CTAs has been completed), then MERLOT measures the current time $t_{current}$. It then calculates how far ahead of schedule it is $t_{ahead}$ by subtracting $t_{current}$ from the worst case time for this checkpoint $wCTA_j$. (Although beyond the scope of the thesis, this interface could easily be extended to detect when a kernel was behind the supposed worst case schedule and trigger a software interrupt on the CPU.) MERLOT then calculates how much time until the next checkpoint’s worst case deadline $wCTA_{j+1}$. We note that if $t_{ahead} > 0$, then the system resources can be reduced without jeopardizing worst case timing. Thus, $t_{ahead}$ represents the timing slack that hardware transfers from one set of CTAs to the other.

Given this timing information, MERLOT’s checkpoint handler then computes the slowdown $\eta$ that is permissible given the timing slack $t_{ahead}$. Note that this slowdown calculation (line 7 of
Algorithm 1) includes the overhead $o$ of changing hardware configurations—i.e., the worst case timing (in milliseconds) of adjusting the number of SMs in use, their frequency and the DRAM frequency—which is set by the hardware developer. $\eta$ is a fixed point number in the range of 0 to 1. Smaller $\eta$ corresponds to more timing slack, meaning the hardware can reduce resources for the next checkpoint.

As a final step, the checkpoint handler transforms $\eta$ into an effective frequency $\lambda$ by multiplying $\eta$ with the GPUs maximum clock frequency $f_{\text{max}}$ and rounding to the nearest whole number. Thus $\lambda$ represents a slower than maximum frequency that will still meet the specified checkpoint deadline. The checkpoint handler then passes this value to the optimizer.

### 4.3 Optimizer

The optimizer takes the effective frequency $\lambda$ and converts it into a frequency for the SMs, a frequency for the DRAM, and the number of SMs to activate for the upcoming checkpoint (inactive SMs are power-gated to save energy). The optimizer here specifies frequencies as integer identifiers. For example, if the system supports 10 frequencies for SMs, then $0 \leq f_{SM} \leq 9$. The resource allocator will convert these identifiers into actual settings for the underlying hardware.

There are two challenges that complicate the optimization process. First, all processors (CPUs, GPUs, and other specialized accelerators) have some critical threshold beyond which additional slowdown actually causes higher energy consumption [48], so the optimizer must avoid reducing resources to the point that energy consumption is actually worse. Second, different applications will need different resources; some will require more memory bandwidth, while others will require more compute. The optimizer must deliver the resources that the application actually needs so that it can save energy.

Regarding the first challenge: while reducing the hardware resources in use will always reduce power, there is an critical frequency threshold $f_{\text{crit}}$ beyond which additional slowdown will actually increase energy consumption [48, 57]. This threshold arises from the fact that chip power
Algorithm 1 Hardware Real-time Management

Require: $c_j[wCTA_j, nCTA_j]$ \hspace{1cm} \triangleright WCET for checkpoint $j$

Require: $t_{current}$ \hspace{1cm} \triangleright current elapsed time

Require: $ALU_{utilization}$

Require: $f_{max}$ \hspace{1cm} \triangleright maximum id for frequency scaling

Require: $f_{crit}$ \hspace{1cm} \triangleright frequency id that gives best energy

Require: $th_{ALU}$ \hspace{1cm} \triangleright ALU threshold that split memory and compute phase

Require: $CTA_{current}$ \hspace{1cm} \triangleright current number of CTA that already finished

Require: $o$ \hspace{1cm} \triangleright Overhead

1: procedure INITIALIZATION
2: $j \leftarrow 0$

3: procedure CHECKPOINT HANDLER
4: if $nCTA_j = CTA_{current}$ then
5: $t_{ahead} = wCTA_j - t_{current}$
6: $t_{remaining} = wCTA_j+1 - wCTA_j$
7: $\eta = \frac{t_{remaining} + t_{ahead}}{t_{remaining} + t_{ahead} + o}$
8: $\lambda = \lceil f_{max} \cdot \eta \rceil$
9: $j \leftarrow j + 1$

10: procedure OPTIMIZER
11: if $\lambda < f_{crit}$ then
12: if $ALU_{utilization} < th_{ALU}$ then
13: $f_{SM} \leftarrow f_{crit} - 1$
14: $f_{mem} \leftarrow f_{crit}$
15: else
16: $f_{SM} \leftarrow f_{crit}$
17: $f_{mem} \leftarrow f_{crit} - 1$
18: else
19: if $ALU_{utilization} < th_{ALU}$ then
20: $f_{SM} \leftarrow \lambda$
21: $f_{mem} \leftarrow \lambda + 1$
22: else
23: $f_{SM} \leftarrow \lambda$
24: $f_{mem} \leftarrow \lambda$
25: $nSM = 0$
26: for all $sm$ in the GPU do
27: if $sm ==$ active then
28: $nSM = nSM + 1$
29: procedure RESOURCE ALLOCATOR
30: if $f_{mem} > f_{max}$ then
31: $f_{mem} \leftarrow f_{max}$
32: Allocate($f_{SM}, f_{mem}, nSM$)
consumption consists of both a dynamic and static component. Dynamic power is decreased with decreased resource usage. While static power will remain constant if only frequency is changed. The critical frequency threshold is the point at which additional slowdown increases energy consumption. If $\lambda$ is below this value, then it is more energy efficient to run the processor at $f_{crit}$ and then transition to a low-power idle state than it is to slowdown beyond $f_{crit}$.

Regarding the second challenge: prior work shows that memory- and compute-bound kernels can be distinguished by both their progress through the hardware scheduling queue and through their arithmetic and logical unit (ALU) utilization [61]. The exact threshold $th_{ALU}$ for distinguishing these two types of kernels is hardware dependent; i.e., it is a function of the SM core and memory design. Therefore, it must be determined by the hardware designer. On our experimental system, we determined this threshold to sit at 20% utilization, which is consistent with prior work [61].

MERLOT’s optimizer uses these two observations to turn the effective frequency $\lambda$ into an actual set of resources for the kernel to use, specified in pseudo-code in Algorithm 1, lines 10–28. Using nested conditionals, the optimizer accounts for four cases in lines 11–24. The cases correspond to whether or not the effective frequency $\lambda$ is below the critical frequency $f_{crit}$ and whether or not the kernel is memory-bound or compute-bound. For the SM frequency, if $\lambda < f_{crit}$, MERLOT simply uses $f_{crit}$, otherwise using $\lambda$. To set DRAM frequency, MERLOT relies on the observation that memory-bound applications should have DRAM set faster than the processor, while compute bound applications should have the DRAM set slower than the processor [61].

After adjusting SM and DRAM frequency, MERLOT sets the active number of SMs in Algorithm 1, lines 26–28. Power gating in our test system is not implemented well, so it is always most energy efficient to use all SMs, as long as there are CTAs to schedule. If there are not enough CTAs to saturate all SMs, then MERLOT records them as inactive, so that the resource allocator can set them to their lowest frequency and save energy. On a system with better power-gating, MERLOT could use SMs as another configuration parameter to tune the performance/energy tradeoffs
Table 4.2: Resource Allocator Translation

<table>
<thead>
<tr>
<th>SM frequency</th>
<th>DRAM frequency</th>
<th>Normalized Voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_{SM} )</td>
<td>( f_{mem} )</td>
<td></td>
</tr>
<tr>
<td>7 MHz</td>
<td>7 MHz</td>
<td>924 MHz</td>
</tr>
<tr>
<td>6 MHz</td>
<td>6 MHz</td>
<td>792 MHz</td>
</tr>
<tr>
<td>5 MHz</td>
<td>5 MHz</td>
<td>660 MHz</td>
</tr>
<tr>
<td>4 MHz</td>
<td>4 MHz</td>
<td>528 MHz</td>
</tr>
<tr>
<td>3 MHz</td>
<td>3 MHz</td>
<td>396 MHz</td>
</tr>
<tr>
<td>2 MHz</td>
<td>2 MHz</td>
<td>264 MHz</td>
</tr>
<tr>
<td>1 MHz</td>
<td>1 MHz</td>
<td>132 MHz</td>
</tr>
</tbody>
</table>

...to meet the required effective frequency with lower energy.

One issue to note is that misclassifications may occur, but they will not cause the system to miss a deadline. The result of a misclassification is that energy efficiency will be lower than optimal. Misclassifications only affects energy (rather than timing) because the optimizer never reduces frequency below that determined by \( \lambda \) in the checkpoint handling procedure; i.e., the frequency is always above the “safe” zone.

4.4 Resource Allocator

The resource allocator translates integer \( f_{SM} \) and \( f_{mem} \) from the optimizer to actual SM and DRAM frequency values and a voltage for each. This step is entirely hardware dependent. The hardware designer has to supply a table mapping the integer identifiers into actual frequency and voltage values. As an example, Table 4.2 shows this mapping for the evaluation system used in this thesis. Lines in 30–31 check whether the frequency identifiers are out of bounds, which can only happen due to the increase in memory frequency in line 21.

4.5 Using MERLOT for Dynamic Timing Analysis

As mentioned earlier, MERLOT can not only enforce deadlines, it can be used to help developers understand how to set those deadlines in the first place. As shown in Algorithm 1, observes the completion time of each CTA before reconfiguring the GPU. Thus, at the cost of an additional...
register for storage, we can easily operate MERLOT in a profiling mode, where it simply updates a register with the longest completion time measured for each CTA (and the GPU resources are not changed). When the kernel completes, software on the CPU can query this register to get the measured CTA time. This profiling mode would be especially helpful during system design and development (before deployment). Developers can run all key kernels in profiling mode and record their observed CTA completion time. Software developers could even stress the system; for example, flushing all GPU caches and memories then run the kernels and measure CTA time using MERLOT’s profiling ability. We emphasize that this support is only for dynamic timing analysis: it reports the worst measured time, but provides no guarantees that it finds the true worst case timing. Developers requiring true worst case timing bounds will need a static timing analysis tool.

### 4.6 Hardware Implementation

MERLOT is implementable in hardware. We do not have the capability to synthesize a GPU that includes MERLOT, but we believe it is important to show that MERLOT can be implemented in hardware. We therefore implement MERLOT in VHDL and synthesize for an FPGA.

To get some specific numbers, we synthesize MERLOT’s VHDL implementation for a CYCLONE-4E FPGA using Quartus II. We use a fixed point package to perform multiplication and division in VHDL [8]. We synthesize the design and find that MERLOT requires 1,048 logic elements, 1 divider and 50 registers. The TimeQuest timing analyzer shows that MERLOT’s $f_{max}$ is 6.61 MHz or 106 cycles overhead in GPU SM frequency. The PowerPlay Early Power Estimator shows that MERLOT needs 0.299 Watts to operate.

The FPGA implementation is not proposed to give an actual implementation of the hardware but rather to show the feasibility of MERLOT’s design. We use GPGPU-Sim [5] as a simulator and GPUWattch for energy measurement [10]. For the experiments, we use the GTX 480 model provided by those simulators. Changing the frequency requires 512 cycles of overhead based on

---

1. While old, no more recent open-source simulator is available.
published models [36]. We include all of this overhead in our reported results.

4.7 Real-Time Considerations

We analyze MERLOT’s hard real-time guarantees by following the example in [53]. This work shows a real-time scheduler guarantees that tasks will meet the deadline if the task set is schedulable. The task set is schedulable under EDF (earliest deadline first) if \( C_1/P_1 + C_2/P_2 + \ldots + C_n + P_n \leq 1 \), \( C \) is wcet and \( P \) is period. Then the task is schedulable under scaled value \( \eta \) if \( C_1/P_1 + C_2/P_2 + \ldots + C_n + P_n \leq \eta \). Line 7 of Algorithm 1 reflects this schedulability test.
CHAPTER 5
RESULTS

5.1 Meeting Hard Deadlines

Determining WCET in GPU still an open problem which currently no exact formula to compute the exact WCET. Therefore for each benchmark we evaluate two different timing profiles, where the worst case timing is a factor of $X$ times the average case timing when the application is allocated all resources and $X \in \{1.5, 2\}$. We note that these relative differences between the average and worst case timing are tighter than found in prior work [6]. We examine the sensitivity to these deadlines in Section 5.3.

The race-to-idle approach only takes advantage of dynamic power management, completing all work as fast as possible and then switching to the low-power idle state. Meanwhile the software approach will slowdown the configuration between the kernels, using a combination of slack transfer, dynamic voltage and frequency scaling and dynamic power management. This software scheduler runs as slow as possible such that the deadline is met by reducing the frequency of SMs and DRAM. MERLOT runs Algorithm 1 whenever the checkpoint is triggered. MERLOT is configured so that each application has 16 checkpoints.

5.1.1 Performance

Figure 5.1 shows how much faster MERLOT is done ahead of the worst case deadline (note the last set of bars on the x-axis is the geometric mean across all applications). Both race-to-idle, software and MERLOT never miss the deadlines. MERLOT in average runs 13.6% faster than the deadlines. We highlight several examples that illustrate the different behavior of the different schedulers.

As discussed in section 3, backprop has 2 kernels each of which take almost equal time, the first kernel is 11% longer than the second kernel. Software cannot configure the resources until the first kernel is finished, but MERLOT configures the resources at every checkpoint. Thus, MERLOT
Figure 5.1: Running time normalized to deadline. The left figure shows the results where the worst case is $1.5 \times$ the average case and the right side shows the case where the ratio is $2.0 \times$.

Figure 5.2: Energy normalized to the race-to-idle approach. The left figure shows the results where the worst case is $1.5 \times$ the average case and the right side shows the case where the ratio is $2.0 \times$.

is closer to deadline—only 5% faster than necessary in 2.0X of ACET, while software is 43% faster than deadline. As we will see, this means that software incurs a higher peak power and higher total energy consumption.

hotspot is an example of an application with a single kernel. Race-to-idle and slack transfer are equivalent for this application because software cannot transfer slack in this application. Meanwhile, MERLOTT can transfer the slack between CTA and slowdown the runtime to save power.

Because it is composed of many different kernels the software slack transfer approach has many chances to slowdown the bfs application. For this application, software actually drops the frequency below the critical threshold (see Section 4.3), which increases energy consumption. MERLOTT’s hardware-based approach is aware that it is not beneficial to slow down this much. Therefore the software approach is closer to the deadline—only $1.42 \times$ faster—and MERLOTT is faster, but saves more energy as we will see in the next section.
5.1.2  Energy Saving

Figure 5.2 shows race-to-idle, software and MERLOT’s energy consumption normalized to race-to-idle (note the last set of bars on the x-axis is the geometric mean across all applications). On average, software spends 93.4\% energy and MERLOT saves 16.73\% energy compared with race-to-idle with 16.43\% in 1.5X and 17.03\% in 2.0X. MERLOT consistently gives a better energy savings than the race-to-idle and software approaches. The main reason is that MERLOT simply has more opportunities to reduce resource usage. This fact is illustrated by the backprop and hotspot applications for which MERLOT provides 20.6\% and 15.4\% energy savings, respectively. In contrast, the software approach reduces backprop’s energy by just 1\% with no savings for hotspot, as it has just a single kernel.

In previous subsection we saw that software approach is closer to deadline than MERLOT for the cfd and bfs applications. This timing occurs because the software is not aware of the critical frequency threshold and slows down the applications too much. This slowdown reduces peak power consumption, but actually increases energy use. The software approach uses 3.5\% and 1.7\% more energy in 1.5X and 2.0X for cfd than race-to-idle, while MERLOT saves 13.99\% and 16.4\% energy. In bfs, software saves 10.19\% and 11.83\% energy, which is worse than MERLOT which saves 17.68\% and 15.05\% energy.

5.2  Comparison of Hard and Soft Deadline

GRAPE is a prior hardware approach to minimize energy in soft real-time systems [57] using adaptive control. GRAPE is based on control theory and cannot guarantee deadlines, but instead achieves close to the desired performance on average. As published numbers show, GRAPE will fail to meet deadlines if kernels within an application vary significantly in achievable performance. Another important difference is that GRAPE tracks application process using instructions retired, while MERLOT uses CTAs as checkpoints. Due to the different guarantees and progress metrics, the user interfaces are also different. GRAPE accepts a single number from a user: the target
rate of instructions per second for a complete GPU application. In contrast, MERLOT requires each kernel to specify deadlines (in terms of number of checkpoints and worse-case timing for those checkpoints). This interface also affects the implementation, as GRAPE periodically checks progress and predicts whether or not the application is on schedule. In contrast, MERLOT triggers only at user-specified checkpoints and performs far less work, as it simply transfers any slack from the previous checkpoint to the next. This difference in workload can be seen in the implementation details: GRAPE’s FPGA implementation requires 18K logic elements and 63 multipliers, where MERLOT requires only 1K logic elements and no multipliers (see Section 4.6). Figure 5.3 compares the performance of MERLOT to GRAPE. Both target twice the default latency. Figure 5.4 compares the energy savings between those two. MERLOT reduces energy by transferring the slack of the task meanwhile GRAPE takes an average of the performance over time using a feedback control system. Figure 5.3 shows that while MERLOT never misses a deadline, GRAPE misses deadlines in all of the benchmarks. While MERLOT provides hard real-time guarantees, GRAPE’s energy saving is slightly worse to MERLOT. MERLOT saves 16.38% energy consumption while GRAPE saves 15.96% energy saving on average. This is because MERLOT already know what is the best frequency to working on $f_{crit}$ while GRAPE is oscillating to find the best energy configuration.

![Figure 5.3: MERLOT and GRAPE Performance](image-url)
5.3 Sensitivity Analysis

We evaluate the sensitivity of our results to 3 factors: (1) the deadline, (2) the underlying architectural support for energy efficiency, and (3) the static (or leakage power). To address the first factor, we simply extend the deadlines to include those that are 5, 10, and $15 \times$ the worst observed latency. To address the second factor we reconfigure the simulation infrastructure to incorporate modern GPU energy savings techniques. Specifically, we emulate a GTX 580 with support for extremely low-power idle states. To adapt the model, we follow established precedent and extend the architecture to 16 SMs, extend the maximum SM frequency to 772 MHz, raise the maximum DRAM frequency to 1002 MHz, and use power models for the GTX 580’s 40 nm technology [49]. We assume a completely linear relation between the frequency and voltage scaling as used in prior work [61]. In addition, we incorporate low-power idle modes; using the most recent NVIDIA technology, it is possible to idle the GPU at just 6.8W which we use as the idle power [3]. We address the final factor by exploring two different values for static power in the GTX 580, the reported value of 41.9W and double this value to a speculative value of 83.8W.

Table 5.1 shows MERLOT’s energy savings over race-to-idle for different deadlines, architectures, and leakage power on average. Figure 5.5 shows MERLOT’s energy saving in 5X, 10X and 15X of WCET with energy saving of 10.74%, 6.91% and 5.19% respectively. MERLOT always saves energy compared to race-to-idle. However, as the deadline becomes a large multiple of the worst case latency, MERLOT’s energy savings advantage starts to diminish on the GTX 480. This
Table 5.1: Average Deadline vs Energy Savings.

<table>
<thead>
<tr>
<th>Deadline</th>
<th>GTX 580 41.9W</th>
<th>GTX 580 83.8W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>33.93%</td>
<td>29.76%</td>
</tr>
<tr>
<td>2</td>
<td>34.92%</td>
<td>30.61%</td>
</tr>
<tr>
<td>5</td>
<td>30.09%</td>
<td>26.98%</td>
</tr>
<tr>
<td>10</td>
<td>24.54%</td>
<td>23.08%</td>
</tr>
<tr>
<td>15</td>
<td>20.89%</td>
<td>20.31%</td>
</tr>
</tbody>
</table>

The effect is caused by the fact that as the deadline is extended a greater portion of the total energy consumption for both MERLOT and race-to-idle is dominated by the hardware’s idle power and the GTX 480 has relatively large idle power. In contrast, we see that MERLOT achieves considerable energy savings on the GTX 580 across all latency targets. This effect arises from the fact that the more energy savings features are available from the underlying architecture, the more opportunity MERLOT has to save energy. Furthermore, with low-power idle states, the energy is dominated by the energy spent executing the task. Race-to-idle performs this computation in a very inefficient state (in terms of operations completed per Watt), while MERLOT performs this computation in the most energy efficient state that still meets the deadline. Essentially, MERLOT implements the pace-to-idle heuristic that Kim et al. show to be uniformly no worse than race-to-idle in all cases [34].

Finally, we see that higher leakage power lowers the relative energy savings, but even when the leakage power doubles MERLOT can still save significant energy in the GTX 580. This is because of the optimization from Algorithm 1 in lines 12-17. These operations perform power trimming to high leakage power. In GTX 580 we still implement scaling factor from GTX 480 so high
proportion of static power helps reduce energy in some benchmarks. Overall, these results show that MERLOT’s energy savings are robust to changes in latency, architecture support for energy savings, and leakage power.
CHAPTER 6
RELATED WORK

Heterogeneous systems have become common place because they allow computation to be performed on the most energy efficient processor. These systems include processors with a single-instruction set and different core types as well as general purpose processors paired with accelerators, like GPUs. The importance of such architectures has led to extensive research on their timing and modeling [22, 2]. A great deal of recent research addresses scheduling techniques that guarantee timing while minimizing energy on such heterogeneous systems [47, 29, 69, 40, 41, 62, 54, 39, 24, 56, 43, 27, 52, 15]. A related effort provides energy guarantees while maximizing user-quality of experience [25].

As this thesis concerns GPU resource management, we focus on GPU-related real-time research. GPUs are attractive because they provide high compute power in a small form factor that is appropriate for embedded and automotive workloads. There are several approaches that provide software support for real-time processing on GPUs. Ptasks is a framework for OS management of GPUs that supports scheduling tasks across multiple GPUs [55]. Given a GPU task to schedule, Ptasks selects the “best” GPU on which to run the task, but it provides no timing guarantees. Gdev is another approach for making the GPU a “first-class” object supported by an OS scheduler [33]. Gdev provides fairness and some quality-of-service guarantees, but does not enforce timing.

Kato et al. propose several frameworks combining OS scheduling support for GPUs with timing guarantees [32, 31]. TimeGraph supports priority-based GPU scheduling to ensure hardware isolation of time-critical GPU tasks [32]. RGEM is a user-space approach that supports real-time GPU scheduling, by ensuring the highest priority tasks are assigned to the GPU first [31]. RGEM also breaks long copy operations into chunks that can be pre-empted. Elliott et al. propose GPUSync, which provides real-time scheduling across one or more GPUs and supports sophisticated operations like copying data from one GPU to another [17]. GPUSync has been used as a platform for real-time vision applications like those required in autonomous vehicles [18]. GPES
is another software framework designed to support real-time processing on GPUs by making long-running kernels pre-emptible [68]. While all of these approaches benefit from the fact that GPUs are more energy efficient than CPUs for tasks like computer vision, none of these approaches explicitly address or reduce the energy of the GPUs they manage. Maghazeh et al. do propose an energy-aware technique for scheduling tasks across both CPU and GPU [44], but it is a software technique and suffers from the same drawbacks as other software techniques: specifically, that software has limited opportunities to take advantage of better than worst case behavior for energy savings as described in section 2.

MERLOT differs from this prior work by enforcing software defined timing requirements in hardware. Hardware has more information about how resources are used by a kernel and it can make faster decisions than software. Thus, where software must be conservative to provide hard timing guarantees, hardware has more opportunities to observe execution and reduce resources to save energy when inputs are not worst case. In that sense, the prior software approaches and our proposed hardware approach should be complementary: software systems like GPUSync can focus on orchestrating computation across multiple GPUs and CPUs, while delegating low-level timing and resource management of individual kernels to hardware, which is better equipped to handle these tasks.

Because it works at the hardware-level, MERLOT is similar to prior approaches that guarantee timeliness for various hardware structures. Examples include predictable memory controllers [16, 35], network-on-chip design to support predictable latency [37], and several processor designs that add timing instructions into the instruction set architecture, including PRET [42] and FlexPRET [70]. MERLOT shares the most commonalities with these last approaches as it directly exposes control of timing to software in the form of registers that specify latency requirements for kernels launched on the GPU. The major differences between PRET/FlexPRET and MERLOT are that (1) MERLOT is designed for GPUs rather than embedded CPUs and (2) MERLOT not only delivers predictable timing but does so while tailoring resource usage to reduce energy consumption.
Of the vast array of prior hardware approaches for reduced GPU energy consumption, the two most similar are Equalizer [61] and GRAPE [57]. Equalizer adds hardware resource management to minimize GPU energy consumption, but provides no timing guarantees—actually reducing performance in many cases. GRAPE provides hardware support for soft real-time guarantees while minimizing energy consumption. GRAPE uses a control theoretic design and, in some cases it cannot deliver the requested timing. Unlike these approaches, MERLOT provides hard real-time guarantees and delivers the required performance on applications where GRAPE fails.
CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Conclusion

The computational demands of embedded systems like autonomous vehicles has created a need for GPU and GPU-like accelerators in these systems. While a great deal of work has been done to support real-time processing on GPU-equipped systems, energy management remains a concern. This thesis argues that a simple hardware interface can allow software schedulers to make timing constraints known to hardware. Once those constraints are known, hardware can quickly detect when applications are running ahead of schedule and reduce their resource usage to save energy without violating their timing requirements. To test this insight we have proposed and implemented MERLOT. We find that MERLOT incurs negligible performance, power, and area overhead; but it can reduce GPU energy consumption substantially compared to sophisticated software-only schedulers. While implemented and tested on GPUs, we believe the insights are applicable to a wide variety of hardware accelerators that break software-specified tasks up into smaller hardware-schedulable units.

7.2 Future Works

In this work we described the opportunity for reducing GPU energy consumption while meeting hard real time deadlines. However, there are two main concerns about this work that need a follow up research. First is the problem of determining the WCET in the gpu system and the second is about deploying the work in a real system.

GPUs are proven faster than CPUs in executing a parallel programs, however GPUs are rarely used in a hard real time applications, especially ones with safety-critical requirements. Scheduling hard real-time cannot be done without the applications WCET information. Unlike CPU, determining the worst case execution time in the GPU is problematic. Determining the WCET requires
a knowledge in scheduling algorithms that allocate CPU workloads. However in GPUs, according to [66] "the hardware and software together implement GPU-specific scheduling algorithms that are proprietary, opaque, and can change without notice.” Therefore, determining WCET in GPUs are subject to many pitfalls and dangerous when applied to application with hard real time requirements. Recent research [20] determines the WCET of a GPU application at the compiler level. They separate memory and computation phases in real-time codes, then arbitrates memory phases from different tasks such that only one core at a time can access the DRAM. Their approach will slowdown the GPU because only one core that can access the RAM, meanwhile the GPU has many idle cores. Another research [28] applies the abstract interpretation to GPU L1 data caches to estimate the worst-case L1 data cache miss rate of GPU applications. However the miss rate of GPUs is not deterministic but probabilistic. Therefore it cannot give precise values of the GPU WCET. Then it is obvious that a more general and deterministic approach is needed such that GPU can be implemented in the real system. For the next step to fill the gap of knowledge in finding a deterministic WCET, we will start from determining a probabilistic WCET in GPU [23]. Probabilistic WCET is often used to provide reasonably accurate WCET estimates in early stage of application design as initial placeholder that will be sharpened in each stage of application production.

Once the probabilistic WCET can be determined in GPU, for the next phase, we want to bring our insight from this work and previous work [59, 58] to the application in a real embedded system. Recent AoT (Array of Things) research [51] in dire needs of neural network accelerator to make the machine learning inference faster to meet their real-time constraint. Currently, their system infers a single neural network using Odroid system [11]. While right now it is feasible to run single neural network inference to counting the car, adding multiple new neural networks will require an accelerator such that it can be finished under the time constraint. Also, those new neural networks may be arbitrary added from user will makes a resource competition problem in the system. To handle this we will implement dynamic WCET estimator [46] during the run-time to adaptively schedule the neural networks. From this theses works, it is possible to decompose a
single application into several checkpoints and adjust the resource based on the time slack. While it is difficult to create a custom GPU that can handle hardware checkpointing, we will modify the application such that it will execute and then reviewed every checkpoints. Instead of run-to-completion, we will monotonically increase utility with the length of execution time [45]. This anytime paradigm will ensure that the hard real time requirements are met even if the WCET precision is low. The energy saving can be achieved by adjusting the resource and stopping the application in one of the checkpoints once the utility requirement is met.

In conclusion, our future research will consist of two new things. First, it will be among the first who implement probabilistic and dynamic WCET in GPU. Second, the insight of this thesis that adjusts the resource in each checkpoint will be leveraged into an adaptive scheduler that will review scheduler decisions. It will ensure the timing requirement by implementing anytime paradigm that gives approximate value in each checkpoint such that the hard real time requirement is always met even in low confidence probabilistic WCET. Our future works also will save energy by adjusting the system resource and application in every checkpoints.
REFERENCES


