Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Identifying computing waste in GPUs

Author: Jeffrey Spaan

1st supervisor: Prof. dr. ir. Ana-Lucia Varbanescu
2nd reader: Dr. Andres Goens

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

March 28, 2024

Abstract

Computer scientists have always followed the intuition (and ignorance) that more resources lead to better performance. However, the sustainability challenges in today's ICT domain require us to use existing hardware (and design new hardware) more consciously. Therefore, besides performance, natural resources usage, utilization, and (energy-)efficiency are becoming important aspects in the selection process of new (super)computing systems. To optimize these metrics, we need to accurately model the relationship between application and hardware. With a focus on efficiency, one way to capture this relationship is to quantify the underutilization of systems by applications. We define the effects of such underutilization as *computing waste*.

Measuring the computing waste of a system is nontrivial. Traditional performance and efficiency metrics only report the current quality of the execution, not the *potential quality*. Therefore, in this thesis, we introduce a first definition of *computing waste*, and we define a workflow for identifying such waste in real applications. Our detection mechanism is based on an *empirical sensitivity study*, where we reduce the resources made available to a given application, and observe the performance impact of this platform "shrinking". For both resource reduction and its impact assessment, we use state-of-the-art simulation. By design, our waste-detection workflow is portable to virtually any application, workload, and system where the appropriate performance model and/or simulator are available.

We demonstrate our waste-quantification workflow on GPU systems, using 7 Rodinia [1] kernels. On the GPU architecture side, we reduce the number of SMs, the core clock, and the memory clock using the AccelSim simulator. We observed that 3 out of the 7 kernels contain waste, i.e., we discovered 3 system configurations with fewer resources that can give equal or better performance than the original system. For these, we showed a reduction of 37% of SMs, 47% of the core clock, and 77% of the memory clock are feasible without compromising performance. We further find that reducing resources can also result in an increase in performance (with up to 63% less time and 58% less energy), and that reasoning about combining different types of waste is nontrivial. Finally, we also identify a significant weakness in our approach: its dependency on a potentially flawed system model/simulator may lead to misleading results.

Contents

1

Introduction

The increase in applications' complexity and need for speed have lead to the rapid development and deployment of a diverse spectrum of computing systems, from multi-core CPUs to many-core GPUs and heterogeneous clusters thereof. In search of high-performance, many applications have been ported and tested on such systems, with little care for the efficient and/or effective use of the hardware resources.

Of particular interest in this context are Graphics Processing Units (GPUs). Their use has grown and diversified significantly over the past decade, from traditional graphics rendering to scientific computing and artificial intelligence. To ensure efficient use of these powerful hardware accelerators, it is crucial to measure, analyze, and optimize the performance of workloads running on GPUs. This traditional approach of *optimizing the application for the system at hand* has lead to the design and development of many GPU-specific algorithms and implementations.

However, there are cases where applications have intrinsic properties (e.g., low operational internsity [7] or high branching factors) that limit their ability to utilize the GPU system effectively. In such cases, instead of optimizing the application for the system, one should consider *optimizing the system for the application.* In other words, where traditional performance engineering tries to identify and mitigate bottlenecks in the application, this approach aims to identify where the imaginary bottle is unnecessarily wide. By reshaping the system (the bottle in this example), one could provide the same performance (i.e., debit) without wasting any resources (i.e., glass).

To reshape the system, we must first identify what and how much is *wasted* within the system. *Computing waste* is our proposal for estimating how much is lost by not utilizing the hardware fully. Although we correlate *underutilization* with *waste*, we prefer to use the term *waste* because it indicates the need to reduce unnecessary resources, whereas underutilization implies the need to increase the utilization of those resources.

Identifying and minimizing waste is beneficial for streamlining both applications and systems, as it can reveal different optimization strategies, aiming at maximizing efficiency instead of minimizing latency or maximizing throughput. For example, waste assessment and reduction could lead to using significantly less hardware, amortizing the cost and construction effort of specialized hardware. Specialized hardware is also becoming increasingly accessible, as energy costs are expected to exceed the procurement costs of HPC hardware [2]. Moreover, specialized hardware can also be designed for *shared* applications or algorithms, such as machine learning primitives for HPC or encryption algorithms for data centers. Lastly, the cost or the number of resources is not always correlated with performance. To maximize performance, minimize cost, and reduce environmental impact, one should identify and minimize unused or inefficiently-used resources.

Waste can be minimized either: *a)* **statically**, through application-system co-design, where the hardware is tailored to the application and/or workload (and vice versa), or simply with a selection between different systems, or *b)* **dynamically**, through throttling, where hardware is reduced on-the-fly, e.g., reducing the core clock when compute operations are sparse, or through co-location, where multiple, simultaneously executing, applications balance resources in order to minimize the overall waste.

In the reminder of this work, we focus on identifying (and possibly) reducing waste *dynamically.*

1.1 Research Questions and Approach

This work aims to demonstrate how to *detect* and *interpret* waste in computing. For this work, we limit our scope to GPUs as complex, representative platforms that are very popular for many workloads, including data-science, scientific computing, and AI. However, we expect that waste exists in every part of the computing continuum, from a single-CPU or heterogeneous system to multi-GPU and multi-node setups, and we strive to provide a methodology that is portable to all these different types of platforms.

Thus, we formulate our main research question as: **What is a functional workflow for measuring and interpreting waste in GPU-based applications?**

To answer this question, we propose a first-of-its-kind architecture for such a workflow, provide a prototype implementation, and validate its feasibility through a set of case-studies. Specifically, we use four GPU-only applications from the Rodinia [1] benchmark suite to demonstrate waste, and, cautiously, reflect on its source and overlap between applications.

We further analyze our results to *a)* differentiate *types of waste* and investigate how they interact, *b)* discover *waste trends* across applications, and *c)* analyze the *effect of workload size* on (system) waste.

1.2 Thesis Outline

The thesis is organized as follows. In chapter 2 we explain the inner workings of GPUs and current methods of measuring and improving performance. We further define, in chapter 3, waste in relation to performance and investigate potential techniques to predict, and minimize waste. In chapter 4, we define our workflow for predicting waste using simulation, and highlight the benefits and challenges of this approach. We propose an empirical evaluation methodology for the proposed workflow in chapter 5, and we analyze our findings to identify and investigate waste. in chapter 6. Finally, we discuss our findings, contributions, limitations and future work in chapter 7.

2

Background and related work

In this chapter we provide the necessary background information for the reader to understand the remainder of this work. Specifically, we introduce the architecture of GPUs, the basics of GPU applications, and methods for GPU performance assessment and improvement. We rely on the terminology and specifics of NVIDIA GPUs, which we use for further analysis and validation. However, the same principles apply to all GPU architectures: they feature hundreds of physical cores, grouped into up to tens logical cores, with a hierarchical memory system and very efficient hardware scheduling for latency hiding. We include a detailed discussion on performance because of our definition of waste, which is strongly correlated to performance and utilization, as seen in chapter 3. We conclude this chapter with a brief summary of relevant related work.

2.1 The GPU architecture

Graphics Processing Units (GPUs) are highly parallel and programmable accelerators. They have evolved from specialized hardware designed primarily for rendering graphics to tackling a wide range of computational tasks, i.e., general-purpose GPU computing (GPGPU). For example, GPUs are used for scientific simulations, artificial intelligence, and cryptocurrency mining. GPUs are also known for their energy efficiency. They provide significantly advantages compared to traditional CPUs [3]: higher throughput, higher energy efficiency (and, therefore, lower environmental imapct), and better cost-per-FLOP.

GPUs excel at executing a large number of threads concurrently, making them throughput-driven and well-suited for workloads that involve vast amounts of parallelism. GPUs use a Single Instruction, Multiple Threads (SIMT) execution model, where a single instruction is executed across multiple threads simultaneously. While SIMT is very efficient for data-parallel tasks, divergent control-flow and synchronization can limit its effectiveness.

To achieve their high-throughput, GPUs feature many, simple, specialized cores. GPUs work as *accelerators*, meaning that they are managed by a host CPU. It is typical that CPU's heavy workloads are offloaded to be accelerated by the GPU, but workload sharing between the CPU and GPU is also possible (often referred to as *heterogeneous computing*). Modern GPUs are highly programmable, supporting high-level languages like CUDA, SyCL, HSA, etc. This flexibility empowers developers to attempt the use of GPUs for wide range of applications.

The GPU components of interest in this work are briefly described in the following paragraphs.

Cores and Streaming Multiprocessors (SMs)

The fundamental computational units of the GPU are *the cores*. Cores are grouped into streaming multiprocessors (SMs), which are comparable, conceptually, to the CPU cores. Each SM contains a certain number of integer, single-precision, double-precision, tensor, and/or ray-tracing cores. The exact numbers depend on the GPU model and generation. Each SM also includes a Level 1 (L1) instruction and data cache (which includes shared and texture memory), and a certain number of warp schedulers. These schedulers ensure fast in-hardware scheduling, enabling very efficient massive parallelism. The number of cores per SM and the number of SMs in a GPU significantly influences its processing power, as more SMs can produce a higher level of parallelism.

From cores to threads

On the programming side, a GPU application is implemented as a (large) collection of threads, grouped into blocks. Blocks are mapped onto SMs, while threads are mapped onto cores. Because all threads in a thread block are executed in the same SM, and because synchronization primitives only span a single block, SM's are inherently scalable.

When transiting from software to hardware, thread blocks are subdivided into groups called warps, each comprising 32 threads. Every cycle, warp schedulers can schedule multiple warps on the actual cores. All threads in the warp execute *in lockstep* - i.e., they execute the same instruction and follow the same control-flow. Overall, warp schedulers manage the execution of warps within SMs: they select and dispatch warps that are ready to execute, ensuring that the SM remains busy by efficiently switching between ready-to-execute warps and avoiding stalls. This mechanism helps the GPU hide high memory latency and maintain high throughput.

The memory hierarchy

GPUs feature a multi-level memory hierarchy., including registers, two levels of cache, specialized memories (such as shared or texture memory), and a global (device) memory. Registers provide threads with private workspace for temporary data. Shared memory enables threads to share data between threads in the same thread block. Global memory serves as the main memory storage on the device, being the largest, but slowest memory level; it provides storage for data accessible to all threads (across all SMs). Unseen by the programmer, memory first travels through the L1 and L2 caches before reaching main memory (and the other way around). This cache hierarchy ensures low latencies for acess patterns with high locality - i.e., applications that repeatedly access the same or close-to-each-other data.

Clocks

WIth their complex architecture, GPUs features multiple clocks, which define the operational "speed" of the SMs and the GPU memory. The *core clock* defines the time per cycle in the SMs (including the L1 caches) and the L2 cache. A higher core clock therefore means the GPU issues instructions faster, performs computations faster, and loads/stores data from/to the L1 or L2 caches faster. The *memory clock* on the other hand defines the time per cycle in the main memory. Increasing the memory clock frequency will results in faster loads and stores to and from the main memory.

Peak performance

The peak performance of computing systems in general, and of GPUs in particular, is calculated as the performance achieved when the system works at full clock-speed using all its resources. As such, for a GPU, the peak performance is achieved when all the cores in all SMs work concurrently, at full clock speed.

2.2 GPU Performance

2.2.1 Measuring performance

Performance can be measured using different metrics, from execution time to energy efficiency or utilization. Measuring execution time can be as simple as running the workload (i.e., the application and the dataset) on the hardware itself, and measuring how long the run takes.

For other performance metrics, such as the power consumption, one can either continuously poll the GPU's overall statistics, for instance using nvidia-smi[1], or by using a profiler such as Nvidia's Nsight Compute[2] or Nvidia's Nsight Systems[3] in order to have more flexibility over the selection of metrics and the sample rate. Other metrics such as floating-point operations per second (FLOPs) or domain-specific metrics such as frames per second (FPS) might require a combination of profiled, measured, and application-specific metrics. However, so long as the system exists, measuring performance is feasible.

For the case where the hardware is not available, various models and/or simulators are available to estimate the expected performance for the given workload.

2.2.2 Improving Performance

When programming GPUs, it is often the case that the observed performance is significantly below the peak performance, which is considered inefficient. To improve efficiency, various performance optimization strategies are employed. We present the most relevant ones in the following paragraphs.

Algorithm optimization

The first step to improve performance is designing or selecting algorithms that perform the least amount of work, or perform the same work more efficiently. This often involves exploiting parallelism and optimizing memory access patterns. For instance, GPU-accelerated libraries like cuBLAS[4] and cuDNN[5] have been handcrafted to harness the full potential of GPU hardware for linear algebra and deep learning tasks, respectively.

[1]`https://developer.nvidia.com/nvidia-system-management-interface`
[2]`https://developer.nvidia.com/nsight-compute`
[3]`https://developer.nvidia.com/nsight-systems`
[4]`https://developer.nvidia.com/cublas`
[5]`https://developer.nvidia.com/cudnn`

Kernel optimization

Profiling and optimizing individual kernels can help identify and eliminate bottlenecks. Techniques include reducing control flow divergence, improving data locality, minimizing global memory accesses, and minimizing data transfers between the CPU and GPU. Profiling tools can measure efficiency and utilization metrics, enabling developers to pinpoint bottlenecks and inefficient segments of the code.

Tuning

: Choosing the right (hyper)parameters for both the kernel and the workload, e.g., `block_size_y` and `elements_per_layer`, can help tailor the application (and the workload) to the hardware. However, traversing this parameter search space by hand or brute-force is infeasible [4]. Fortunately, there exist several tools - such as Kernel Tuner [5] - that can automate and optimize parameters tuning for arbitrary metrics.

Dedicated hardware

: Offloading computations to specialized hardware can improve performance because, arguably, the hardware is inherently optimized for the task at hand. Moreover, offloading can reduce the workload for conventional cores, leading to both lower latencies and more parallelism. As an example, in the context of gaming and real-time graphics, ray tracing technology has improved rendering quality and increased FPS by offloading ray tracing calculations to dedicated hardware, so-called ray tracing cores.

2.2.3 From High-Performance to Zero-Waste

In the remainder of this work, we follow the intuition that high-performance and low-waste are correlated. Intuitively: given the peak performance of a compute system, when an application can perform close to this peak, the hardware resources are highly utilized, and therefore the platform is efficiently used, leading to very limited losses; conversely, when application performance is significantly lower that the (feasible) peak, the platform resources are poorly utilized, and thus the system is poorly utilized, further meaning there exists compute waste in the process.

We acknowledge that poorly written and/or unoptimized applications (see 2.2.2) will lead to high compute waste due to poor software. This *software-induced waste* must be tackled by optimizations that are mandatory for any code in production, as they will increase the efficiency of the software. We further argue there also exists *hardware-induced waste*, where the system simply has too many resources (or unsuitable ones) to be effectively utilized by the workload at hand. Think, for example, of a sequential application running on a multi-core processor: we can intuitively assess that several cores will be "wasted" by this application, but the question how much waste that actually entails still needs an answer. In the remainder of this work, we assume the given code is production-ready and only address the waste quantification from the perspective of the hardware underutilization. By proposing a systematic approach for this quantification, we argue that further optimized software can be easily re-evaluated.

2.3 Related Work

This section lists related work relevant to predicting and understanding computing waste.

2.3.1 Characterizing applications and workloads

Chen et al. [17] analyzed the effect of application characteristics on the power consumption. Their work uses GPGPU-sim [18] to collect more metrics (than benchmarking) from a variety of benchmarks. These metrics are combined in a statistical model which predicts the power consumption. They find that the number of register accesses, the number of single-precision instructions, and the number of main memory instructions are the most indicative for the power consumption.

Abe et al. [19] evaluate impact of the core and memory clocks on a heterogeneous system using the Rodinia [1] benchmark. They confirm the intuition that memory-intensive applications favour a lower core clock in terms of power and not in terms of time.

The work from Kerr et al. [20] proposes a novel set of metrics which they use to analyze applications from the CUDA SDK and Parboil [21] benchmark suites using their own PTX functional simulator. In terms of control- and dataflow, they use the activity factor (the ratio of active threads for a specific warp instruction), divergent branches, and the ratio of data exchange between threads. In terms of memory, they introduce the ratio of operations to main memory *per instruction* and the ratio of main memory transactions *per memory instruction*. In their words, these metrics measure the memory *intensity* and memory *efficiency*, respectively. Lastly, in terms of parallelism, they analyze theoretical SM scaling using the ratio of total number of instructions to the maximum number of instructions per block, with the intuition that increasing the number of SMs is more useful if the amount of work per thread is low.

Che et al. [22] analyze and characterize a set of applications from the Rodinia and Parsec benchmarks. They focus on IPC, active threads, (memory) instruction mix, warp occupancy, shared memory usage, and cache miss rates. Using these characteristics, they group applications using Principal Component Analysis (PCA) and hierarchical clustering. They find that both benchmarks are diverse and complementary.

O'Neil and Burtscher [23] examine the behaviour of applications with irregular memory access patterns and control flow (such as graph and tree algorithms). They characterize irregular control flow behaviour through increased branch divergence, load imbalance, and synchronization overhead from (extended) metrics reported by GPGPU-sim. However, they find these factors are not as limiting to performance as the insufficient bandwidth and high main memory latency caused by the difficulties of regularizing and/or coalescing irregular memory access patterns.

Goswami et al. [24] introduce a set of architecture-agnostic workload characteristics and a set of evaluation metrics to accurately evaluate the design space. The workload characteristics are meant to to capture kernel stress, kernel characteristics (e.g., arithmetic intensity (ai)), divergence characteristics, instruction mix, and coalescing. In contrast, the evaluation metrics are architecture-dependent: memory efficiency (frequency of memory requests), and SIMD parallelism (speedup with infinite threads per SM). They analyze (and cluster) their results for the CUDA SDK, Rodinia, and Parboil benchmark suites.

Carvalho et al. [25] evaluate the Rodinia, Parboil, and SHOC [26] benchmark suites using their selection of characteristics: instruction mix (including per memory type), (achieved) occupancy, and SM efficiency[1]. They cluster their kernels (unlike most related work, where they evaluate on an application level) using PCA and K-means and find four distinct groups: low/medium/high-"resource utilization" and "low occupancy and high efficiency".

Overall, we observe that most papers characterize applications to analyze performance, evaluate the overlap between applications, or for the sake of characterization itself. None of the papers systematically related application and workload characteristics to architectural choices or trends.

2.3.2 (Faster) design space exploration

Jia et al. [12] present Stargazer: an automated regression-based statistical model for efficient design space exploration. At the core, they use a regression model to which they iteratively add architectural parameters in order of (automatically defined) importance to

[1]We are not sure what this characteristic exactly measures. The authors explain it as the "time percentage that at least one warp is active in a SM in relation to all the GPU SMs."

the overall prediction accuracy. The model learns from simulations sampled from the full design space. This allows the model to use 4–5 orders of magnitude fewer simulations than an exhaustive search. Although the model currently only predicts a single performance metric, this could in theory be expanded using multivariate linear regression.

Boroujerdian et al. [27] introduce a design space exploration framework for domain-specific hardware. Their framework is built on top of their simulator "FARSI" which uses (roofline-based) analytical models along with phase-driven[1] simulation to achieve a 8400% speedup over similar simulators. To efficiently traverse the design space, they use the search heuristic Simulated Annealing (SA) enhanced with 'architectural reasoning' (which promotes configurations likely to reduce current bottlenecks).

Liu et al. [28] present a Photon: a systematic methodology to accelerate GPU simulations. Photon uses an online analysis that samples kernels, blocks, and warps to discover if a particular level can be switched from the more accurate, but slower, trace-driven (timing model) to a simpler functional simulation (using the time prediction from the trace-driven sample). Starting from kernel-level, Photon will switch when a kernel/block/warp is stable enough to be representative for other kernels/blocks/warps. Interestingly, they find that (block-level) sampling is even possible for irregular workloads with high levels of divergence and/or synchronization. Their sampling methodology was able to decrease execution time by 99% with a $\sim 10\%$ error compared to the underlying simulator.

Wang et al. [29] present the GPU Memory Divergence Model (MDM) which unlike contemporary analytical models is able to faithfully model memory-divergent operations, i.e., operations with strided or data-dependent access patterns causing loads from different threads to access different cache lines. In the worst-case scenario, concurrent divergent operations can cripple the L1 (and the rest of the memory hierarchy), leading to stalled warps and unhidden latencies. The authors find that these memory-divergent kernels are relatively prevalent (21 of 56 for their benchmark suites). Their model is based on predicting the steady-state performance (with a perfect memory) and subtracting the performance loss caused by cache misses. The authors use a cache simulator to accurately predict the cache losses (for memory-divergent warps). Overall their model is $\sim 600\%$ faster than functional simulation and only $\sim 14\%$ less accurate.

Besides the characterization of applications, as shown in subsection 2.3.1, Che et al. [22] also perform a design space exploration using GPGPU-sim [18]. Their parameters include SIMD width, memory channels, bus width, and core & memory clock. To decrease the

[1]Phase-driven simulation only models the scheduling of statically calculated phases, or time slices, of all 'tasks'.

search space, they employ the Plackett-Burman (PB) [30] design approach to determine the effect of parameters on performance. This method only requires $2n$ simulations instead of 2^n for the full Cartesian product (for n parameters). Unfortunately, besides a few outliers, the authors do not share the results of their experiments. Furthermore, they do not relate the performance between systems to (their) application characteristics.

O'Neil and Burtscher [23] perform a design exploration for applications with irregular memory access patterns. Using GPSPU-sim, they vary the size of the caches, the size of the cache miss queues, the number of miss-status handling registers (MSHR), and the SM-L2 and L2-memory bandwidths. However, their exploration is not very deep, as they only vary 2 parameters at a time. Furthermore, their analysis is focused on relating architectural choices to irregular-memory applications in general, not to (their own proposed) application characteristics (as shown in subsection 2.3.1).

Overall, related work on design space exploration is primarily focused on increasing the accuracy and decreasing the run-time. They do not consider optimizing the architecture for specific applications, application-types (except irregular-memory applications), or algorithms. Furthermore, they mostly focus on accurately predicting performance or, for the related work that actually *explores*, minimizing the performance. Waste, utilization, and (energy-)efficiency are rarely considered metrics of interest.

2.3.3 Optimizing resource-efficiency

Yeung et al. [31] present a statistical model for predicting GPU utilization for different Deep Learning (DL) workloads. Their model is based on the the number of layers (per type of layer), the size of the workload and the number of flops. They also measure the cumulative utilization when co-locating multiple workloads (with the scheduling based on their utilization prediction results). The co-located workloads achieve a 70% utilization while the single workload only achieves 47%. Unfortunately, their model predicts Nvidia's utilization metric which denotes the percentage of time that *a* kernel was executing on the GPU, not the actual utilization of the SMs or the memory.

Bakhoda et al. [32] analyze the effect of reduced resources (and memory controller & interconnect design) for a variety of applications. They find that most applications are more sensitive to the bandwidth than the latency, executing fewer blocks can improve performance by unburdening the memory hierarchy, and that inter-warp coalescing can improve performance by 41%.

Darabi et al. [33] present a technique for reusing unused cores as caches. The technique is based on the intuition that for most kernels, bandwidth bottlenecks performance, meaning that many cores remain idle. Using their technique, each idle core will switch to “cache mode” in which it donates its available register file, shared memory, and L1 cache to a pool of unused caches used by the cores in “compute mode”. For a selection of memory-bound kernels, they observe on average a 39% increase in performance and a 58% increase in energy-efficiency.

Jararweh et al. [34] present an algorithm for determining the optimal number of SMs at run-time in order to save power. In the algorithm, the online scaling of SMs[1] is based on the memory bandwidth utilization. This is based on the fact that kernels that do not share data will scale the bandwidth linearly with the number of SMs. Once the bandwidth saturates, no more performance can be gained from extra SMs. Using the algorithm, the maxtrix multiply application consumed 14.2% less power with only a 1% performance loss.

Li et al. [35] present MISO: a technique for dynamically scheduling co-located applications on (Nvidia’s) Multi-Instance GPU (MIG) partitions (see section 3.2). To know the optimal partitioning at runtime, MISO includes a statistical performance prediction model for MIG partitions, based on the more flexible Multi-Process Service (MPS) configurations which partitions SMs but, unlike MIG, does not partition the memory. The result of the model is used to dynamically determine the optimal partitioning for MIG co-located kernels. Based on a variety of DL workloads, the optimal partitioning achieves a 49% and 16% higher throughput compared to non-shared and statically partitioned kernels respectively.

Park et al. [36] present a technique for dynamically scheduling co-located applications using both MPS[2], which partitions SMs, and simultaneous multikernel (SMK), which allows multiple kernels on the same SM. Like MISO, their scheduler performs a trial of a few partitioning setups to test, on a small scale, the direction of the optimal utilization. On average, their technique is able increase throughput by 45.0% compared to a non-shared workload

Yandrofski et al. [37] introduce a technique for more reliable worst-case execution times (WCETs) measurements using “enemy programs” that intentionally contend for GPU resources. Using MPS and SMK, the enemy programs populate half of the SMs or SM, respectively. The enemy programs then stress a particular resource, e.g., double-precision cores or the L2 cache (using large strided memory loads). Overall, this technique can more accurately measure the sensitivity of a kernel to specific “interference channels” (compared

[1]It is unclear how SMs are scaled dynamically. The authors report “moving unused [SMs] to a low power state”, but the authors do not mention how this is achieved.

[2]The authors refer to this as spatial multitasking because, at the time, it was still a proposed technique.

to only compute-intensive vs. memory-intensive). As such, it can also be seen as design exploration technique.

In general, we observe a high variety of methods for improving resource-efficiency. However, none of the related work performed a systematic analysis of resource-efficiency for specific applications (most papers target every application). Furthermore, most of the related work (implicitly) only targets the utilization of *existing* hardware.

3

Defining Computing Waste

In this chapter we propose a first definition of computing waste, and discuss its implications on the design of our workflow. Specifically, we investigate how waste could be measured, and strategies to limit waste, preferably without hindering performance. Throughout this discussion we use NVIDIA's GPU terminology for consistency, but we note that our definition and approach are in no way platform specific.

3.1 Computing Waste

Waste occurs when *something* is *not used* or *not used effectively.* In the context of computing, waste means an un- or underused part in either the application, workload, or system. There are multiple metrics that can, in different ways, measure this: (under)utilization, efficiency, occupancy, etc. We choose to base our definition of waste on performance. This is because performance, as shown below, is a type of metric that is able to combine the state of the application, workload, and system into a single figure.

Definition 1 (performance)**.** *Performance as a function of the execution:*

$$\text{performance} = f(\text{application}, \text{workload}, \text{system})$$

Here, the unknown function f is the execution (on a GPU). The exact function cannot be defined, because if we could, we would not need GPUs. For our purposes it is only necessary to know that f can transform the inputs into a single performance figure.

One can also subdivide this function further. For instance, for heterogeneous applications, we can write it as follows:

$$\begin{aligned}\text{performance} = \; & f(\text{application}_{\text{CPU}} + \text{application}_{\text{GPU}}, \\ & \quad \text{workload}_{\text{CPU}} + \text{workload}_{\text{GPU}}, \\ & \quad (\text{compute}_{\text{CPU}} + \text{memory}_{\text{CPU}}) + (\text{compute}_{\text{GPU}} + \text{memory}_{\text{GPU}}) + \text{interconnect})\end{aligned}$$

To discover waste, we can reduce one aspect and fix all others. If the performance does not change, or perhaps even improves, we have eliminated waste.

Definition 2 (system waste)**.** *Equal or better performance with a reduced system and a fixed application and workload.*

Finding waste is possible at any level of granularity. At the highest level, we get the following types of waste:

Application waste occurs with a variable application and a fixed system and/or workload. Assuming the design/algorithm(s) of the application are fixed, application waste signals the presence of unnecessary or inefficient computation in the implementation. Finding and eliminating application waste is the primary focus of performance engineering.

Workload waste occurs with a variable workload (in size or form) and a fixed system and/or application. Although the workload is usually considered fixed, i.e., calculating a certain output requires certain inputs, there might still be room to 'optimize' the workload. For instance, to yield roughly similar outputs with potentially less wastefull computation, one can adjust the level of precision, use a different graph format, or use a lower graphics quality in rendering. On a larger scale, a different workload distribution across systems could improve individual utilizations.

System waste occurs with a variable system and a fixed application and/or workload. System waste occurs when a different system is able to execute the application more efficiently. In this case, there are either *unused* or *poorly used* resources (e.g., for GPUs: idle SMs, unfilled caches, an unsaturated memory bandwidth, etc.).

Although our workflow might hold for other types of waste, for the rest of the thesis we will focus primarily on *system waste*, hereafter simply referred to as *waste*.

3.2 Predicting system waste

To reduce waste we first need a way to measure, or, since the target system might not exist, predict, waste. This involves finding a method which can answer the questions: how much, and which parts, of the system will be wasted if I run my application x with workload y on the baseline system z_b instead of the (reduced) system z_i?

To identify the waste, we first need to predict performance. We have identified the following approaches, and summarized them in Table 3.1:

Benchmarking If (a portion of) the systems to compare are readily available (or possible to change dynamically, e.g., clock frequencies), benchmarking is best way to measure waste. Executing and analyzing the application on the target hardware directly should always be preferred over methods that try to approximate the system. Overall, benchmarking is fast, accurate and easy to set up, however, it does require all systems to exist and be accessible (which can be expensive or unfeasible for many-GPU setups, such as data centers or HPC).

Simulation A simulator is a type of model that mimics the execution on the target system in software. Simulation provides a flexible way to experiment with different hardware configurations, programming models, and algorithms without implementing the actual hardware [2].

GPU simulators use either the intermediate/virtual Instruction Set Architecture (vISA) (for Nvidia: PTX) or the machine Instruction Set Architecture (mISA) (for Nvidia: SASS). These simulators are referred to as *execution-driven* and *trace-driven* respectively. The mISA is more explicit and therefore has a more accurate scheduling (and count) of instructions. Simulating the vISA on the other hand can be useful if the simulator needs to be *functional*, i.e., also able to compute the actual output.

In contrast to proprietary simulators used by GPU vendors, open or academic simulators often struggle to precisely model GPU architectures because architectural-level knowledge is either unknown, obscure or too high-level. For example, the authors of Accel-Sim [3] discovered the existence of a previously unknown streaming cache technique in contemporary Nvidia GPUs.

In comparison to other methods, simulation excels at the accuracy and the amount of insight it can provide, at the cost of excessive run-times.

Hardware-assisted simulation Hardware-assisted simulators, sometimes called emulators, typically simulate the target hardware on a lower level than software simulation. They are more difficult to construct as the actual chip design, i.e. the logic gates, have to be imitated. These simulators are usually constructed using field-programmable gate arrays (FPGAs), which provide a speedup of hundreds of thousands times over software emulators, without sacrificing accuracy [2].

Because of the extreme modelling detail and the extensive use of accelerators, hardware-assisted simulators are both faster and more accurate than conventional simulators. However, they are even more difficult to construct and therefore usually limited in scope [2].

Co-location Co-location is the act of simultaneously executing multiple kernels. To measure performance of a system with reduced resources, one could co-locate a kernel that exclusively uses a specific resource, i.e., excessive L2 traffic to reduce the available L2 throughput for the actual application. With the right microbenchmark, any resource can be stressed. However, co-locating a different kernel will inevitably lead to spillage, i.e., inadvertently using other resources. For example, to use the main memory, the kernel must first overflow the L1 and L2 cache.

Besides the problems with implementation effort, accuracy, and scope, co-location excels in speed because it can be performed directly in hardware.

Partitioning Nvidia introduced the concept of Multi-Instance GPU (MIG) [6] which partitions a GPU into multiple devices. Each instance receives the same amount of SMs, a subset of the main memory, and a fixed portion of the bandwidth. Currently, a MIG partition can be composed of up to 7 instances, thereby creating 7 different, previously nonexistent, potential systems. Alternatively, one can also use Multi-Process Service (MPS), which only partitions SMs, not the memory hierarchy.

Although the amount of imaginary ‘systems’ is currently limited, as all resources are equally scaled, future partitioning methods (able to arbitrarily regulate specific resources) could be as fast as benchmarking without the inaccuracies of co-location.

Analytical model Analytical performance models use equations to describe high-level behaviour based on (characteristics of) the source code. Most analytical models, such as the roofline [7] and the Volkov [8] model, compute a form of throughput, which can be transformed into cycles (and therefore time) by counting the number of instructions or operations. The accuracy of contemporary analytical GPU models can vary widely, e.g.,

Technique	Imaginary systems	Setup effort	Run-time	Accuracy	Feedback
Benchmarking	×	Very low	Very fast	Very high	High
Simulation	✓	Low	Slow	High	High
Hardware simulation	✓	Very high	Medium	Very High	Very High
Co-location	✓[1]	High	Very fast	Low	High
Partitioning	✓[2]	Low	Very fast	Very high	High
Analytical model	✓	Low	Instant	Medium	Low
Statistical model	✓	Low	Instant	Medium	Low

Table 3.1: (Dis)advantages of waste prediction methods.

the reported average errors for the Volkov model using the kernels from the Rodinia [1] benchmark vary from 2 to 78%. In comparison to simulation, the state-of-the-art achieved an average error of 10% (to the simulator) [9]. Overall, analytical models are fast and insightful. Furthermore, they are able to explore a much larger area of the design space than can be empirically measured [10]. In comparison to other methods however, analytical models lack the accuracy and variety of reported performance and efficiency metrics. Furthermore, analytical and statistical models, in general, have difficulties predicting irregular workloads where data access patterns only become clear at run-time [11].

Statistical model Statistical models can predict performance by learning from previous executions. By learning, the model will implicitly discover relationships between the inputs (the application, workload, and configurations) and the measured output (performance). Note that in order to train the model, a different model, e.g. a simulator, must be used to create the learning set (as existing hardware is not representative for all hardware) and validate the results [2]. As such, the objective of many statistical models is to limit the number of simulations required to evaluate configurations from a much larger architectural design space [12, 13]. The accuracy of statistical models depends on the underlying performance model and on the quality and diversity of the learning set.

[1]Options are limited by the ability to restrict specific resources.
[2]Options are (currently) limited to specific configurations.

3.3 Minimizing system waste

This section lists the techniques that can be used to minimize or eliminate waste. Interestingly, note that some of the methods used to *minimize* waste can also used to *measure* waste.

Throttling By throttling, i.e., dynamically reducing or completely turning off idle or underused resources, one can create a system with fewer resources, thereby possibly reducing waste. Contemporary GPUs can, using Dynamic Voltage and Frequency Scaling (DVFS), limit the core clock, memory clock, and the power limit, in return for less power consumption [14]. Although the number of resources that can dynamically be reduced is quite small, future GPUs could allow for more customization. For instance, newer CPUs can dynamically schedule the same compute to either faster performance-cores or more energy-efficient e-cores.

Co-location Co-location can be used to leverage wasted resources for other kernels. As an example, combining a compute-bound kernel with memory-bound kernel will could improve the utilization of each resource-type. Minimizing waste using co-location is limited by shared resources, which, for some resources, like SMs, will always occur while for other resources it might happen accidentally if the wasted resources in one kernel are mismatched with oversubscribed resources in the other kernel.

Co-design Co-design is a method of designing software and hardware concurrently in order to maximize a performance metric(s) [15]. If we fix the software side, this means the creation of specialized hardware. Co-design if traditionally used to optimize performance, reduce cost, increase energy-efficiency, etc. However one can also optimize the design to reduce waste, i.e., implementing the optimal system that wastes the least resources, or alternatively, the system that reduces the resources that decrease performance the most (if they would be included). This depends on the balance between goals: reducing hardware and increasing performance.

An example of co-design is the Fugaku supercomputer [16] which uses co-designed CPUs optimized for a collection of applications. On an algorithm-level, examples of co-design include AI accelerators such as Google's tensor processing units (TPU)[1] or graphcore's Intelligence Processing Unit (IPU)[2]. Co-design allows for a deeper control with many more

[1]`https://cloud.google.com/tpu/`
[2]`https://www.graphcore.ai/`

degrees of freedom than general-purpose chips other reprogrammable hardware such as Field-programmable gate arrays (FGPAs) [2]. Therefore, although the process can be prohibitively expensive, co-design could be able to remove even the finest levels of waste.

Selection The most obvious way to minimize waste is to simply procure the system that wastes the least resources. Naturally, a pure selection is only possible if the optimal system does already exist. This scenario can be realistic however if, for example, one had to choose a GPU within a certain generation. GPUs within a generation often differ only by a few resources, e.g. SMs, L2 cache size, and/or the memory clock.

4

Workflow

This chapter defines our workflow for measuring, or more appropriately, *predicting*, single-system, single-GPU, and single-stream waste using simulation, as shown in Figure 4.1.

4.1 From definition to assessment

From section 3.1, recall our definition of (system) waste:

Definition 2 (system waste)**.** *Equal or better performance with a reduced system and a fixed application and workload.*

To asses system waste, we first need know how to reduce a system, and, subsequently, how measure its performance. Our workflow, as shown in Figure 4.1 defines the following steps. First, we 'reduce' a system by reducing the quantity of a preexisting resource. For instance, if the original system has a 6 MB L2 cache, a reduced system might only have a 2 MB L2 cache. Second, if the new system exists, we can benchmark the system to measure the performance. However, for most resource reductions, the system does not exist. We therefore use a performance model. This model must be able to use the application, workload and configuration of resources to produce a performance estimate. Finally, we use the difference between the performance of the reduced and original system to assess whether the original system contained waste.

4.2 Application and workload considerations

Before finding any type of waste, one must first optimize the application and workload. Because, if the goal is to optimize the system for the application, using a different application first (i.e., the underoptimized one) will likely give different and less useful results [11]. The same is true for the workload.

We also calculate several application characteristics in order to discover waste trends across applications in the analysis. For instance, we calculate the arithmetic intensity (AI), i.e., the ratio of compute operations to memory traffic, and use it along with the Roofline model [7] to calculate whether the application is predominately focused on compute or memory.

4.3 System exploration

To predict the performance of *any* system, our workflow requires a performance model. Out of the options listed in section 3.2, we choose simulation because it can achieve the highest accuracy out of all currently available techniques capable of modeling a large, or possibly infinite, variety of GPUs. Unfortunately, simulation is hindered by high execution times. The practical limit of testable configurations is therefore much lower.

4.4 GPU simulators

Where CPU architecture has matured over the years, GPU architecture has been rapidly evolving over the last decades, from changes in the memory hierarchy such as off-chip shared memory and texture caches to computational overhauls such as larger-scope synchronization primitives and specialized hardware like tensor and ray tracing cores. This speed of development has limited the accuracy and lifetime of open-source and/or academic GPU simulators [3], requiring them to be continually updated, highly extensible, and, unfortunately, limited in scope.

Besides proprietary in-house simulators, such as Nvidia's NVArchSim [38], multiple open and/or academic GPU simulators currently exist. The main differences take place in the platform (e.g., AMD, Nvidia, Intel, etc.) and ISA type (e.g., PTX versus HSA(IL)). For the demonstrations of our workflow, we have considered the following simulators:

- **gem5** [39] supports GPU modelling through its (AMD) GCN3-level APU model [40]. Alongside the cycle-accurate Gem5 CPU simulator, it includes an heterogeneous GPU/accelerator model built upon the Radeon Open Compute platform (ROCm) platform[1]. It is capable of executing Heterogeneous Systems Architecture (HSA) [41], OpenCL, and HIP[2] code. HSA is an heterogeneous computing abstraction which

[1] `https://github.com/ROCm`

[2] `https://github.com/ROCm/HIP`

includes unified CPU-accelerator memory (i.e., same address space) and implicit on-demand copying. The APU model is mostly focused on heterogeneous computing and the challenges that come with it, e.g., synchronization and coherency.

- **MGPU-sim** [42] is a multi-GPU/multi-accelerator simulator that supports AMD's GCN3 ISA through their own OpenCL implementation. It is presented as a "high fidelity, high flexibility, and high performance [simulator]" [42], partly due to its parallel execution of different GPU components (memory controllers, compute units, etc.). Besides being able to simulate a single discrete GPU, MGPU-sim can also model both discrete and unified multi-GPU systems.
- **MacSim** [43] is a heterogeneous architecture simulator. In terms of GPUs, it is able to simulate Nvdia's Fermi and Intel's GEN9 GPU architecture [44]. MacSim can be both trace- and execution-driven, depending on the used ISA.
- **Multi2-sim** [45] is a heterogeneous functional and architectural simulator. Execution-driven GPU simulation for AMD's Evergreen architecture is supported through their own OpenCL implementation. Furthermore, Multi2-sim also supports a trace-driven simulation for Nvidia's Kepler architecture [46].
- **Accel-Sim** [3] is an Nvidia-based trace-driven GPU simulator. It also supports execution-driven PTX simulation through its underlying GPGPU-Sim [18] simulator. Accel-sim is the only simulator capable of simulating arbitrary (CUDA) kernels because of its (mISA) SASS support. Accel-Sim also incorporates a power model, AccelWattch [47], based on the McPAT [48] modeling framework.

For our experiments, we opt for Accel-Sim because it (i) has an integrated power model, (ii) is the most accurate out of all publicly available GPU simulators [3], and (iii) is in active development and therefore more future proof. However note that our workflow is applicable to any GPU simulator (or performance model).

4.5 Steps to reproduce

The main steps to reproduce our results with Accel-Sim are as follows:

1. Trace the application and workload using Accel-Sim's tracing tool through Accel-Sim's `run_hw_trace.py` script.

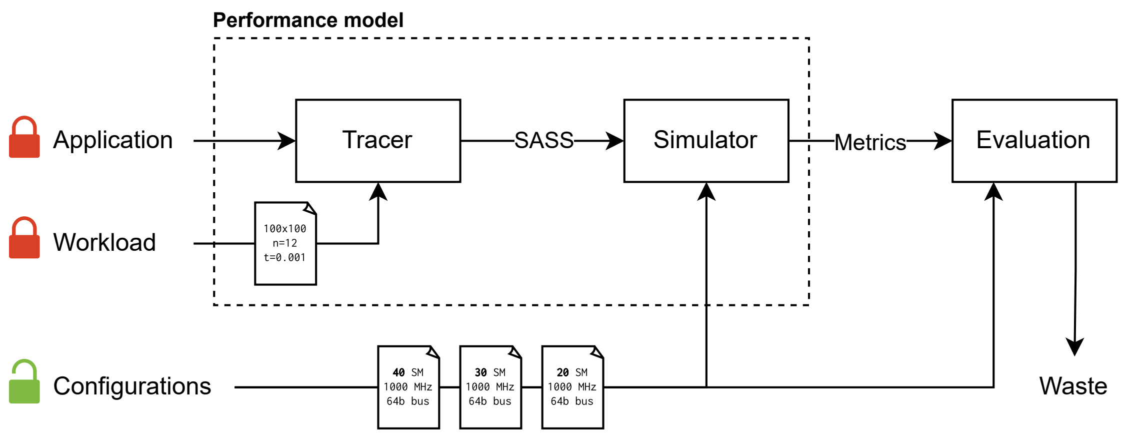

Figure 4.1: Our workflow for measuring waste in GPU-based applications.

2. Define a machine configuration (in a `gpgpusim.config` file), or alternatively, alter one of the default, pre-validated, configurations. Accel-Sim provides tested configurations for the GTX480, Titan, Titan X, RTX 2060 RTX 2060 Super, Quadro V100, Titan V and RTX 3070 GPUs.

3. Run the simulator with the trace and the machine configuration. Accel-Sim can be executed using the `accelsim.out` binary, or with the provided launch script `run_simulations.py`.

4. Collect Accel-Sim and Accelwattch metrics. Accel-Sim provides a `get-stats.py` script to easily aggregate the results across kernels and kernel invocations.

5. Calculate composite metrics such as execution time, instructions per Joule, etc.

6. Using definition 2, analyze the results to discover (trends in) waste.

4.6 Collecting metrics

Instead of looking for utilization metrics (e.g., memory bandwidth, pipe usage, etc., versus the theoretical peak), we infer waste from performance metrics. These metrics however can measured at different levels of granularity. For example, we can (i) aggregate the entire application, (ii) use a per-kernel aggregate, or (iii) use a per-invocation aggregate. Option (i) is the most realistic, but prone to 'hide' waste in its aggregates. With enough kernels, every resource will inevitably be used by one of them, meaning that the overall optimal system will most likely be the system with the most resources (even though all but one kernels might not use that specific resource).

The ability to discover waste is much higher with option (ii) and (iii) because waste across kernels, or invocations respectively, is distinct, i.e., different kernels/invocations can have different waste. Eliminating waste becomes a bigger challenge for (ii) and (iii) however because each kernel/invocation requires specialized hardware, which can, especially for the invocations, quickly increase construction cost and effort. Moreover, most kernels exhibit the same behaviour across invocations. Analyzing each invocation separately can therefore be a fruitless venture.

For the reasons above, we chose to aggregate performance on a kernel-level for our demonstrations. However, note that our workflow does not dictate the approach. We leave this up to the user.

To find waste we record and/or calculate the following metrics for each kernel:

- **Execution time**, calculated as $\frac{\text{simulated cycles}}{\text{clock frequency (hertz)}}$.
- **Energy** in Joules, calculated as average power (Watt) $\times$ execution time (seconds). Note that the maximum power will be capped in hardware.
- **Instructions per cycle (IPC)**, calculated as $\frac{\text{simulated instructions}}{\text{simulated cycles}}$. Note that we count instructions at thread-level, not warp-level ($\times 32$).
- **Energy efficiency** in instructions per Joule, calculated as $\frac{\text{simulated instructions}}{\text{energy (J)}}$. Accel-Sim does not record the number of floating point operations, so instead of the commonly used flops/s per watt we use instructions per joule. Note that these metrics are comparable: $\frac{\frac{\text{ops}}{\text{s}}}{\text{watt}} = \frac{\text{ops}}{\text{watt}\cdot\text{s}} = \frac{\text{ops}}{\text{Joule}}$. Although operations are not synonymous to instructions, we assume both metrics roughly exhibit the same behaviour.

Although ultimately up to the user, we use execution time and energy as the foremost metrics to judge the quality of the systems. As such, we hereafter refer to them as *performance* metrics. We use instructions per cycle and instructions per Joule to reason about efficiency.

5

Experimental setup

In this chapter we provide the details about our experimental setup. Specifically. we present the hardware platform (and simulator) we used, and the applications we selected as case-studies.

5.1 Platform and simulator

The baseline system is a Nvidia RTX 2060 Super, which we chose because it is the newest generation card validated in the Accel-Sim paper and repository [3]. We follow Accel-Sim's given configuration of the RTX 2060 which has 34 SMs, a core clock of 1905 MHz, and a memory clock of 3500 MHz. Note that the actual frequencies can vary between retailers[1]. For instance, Nvidia reports a 1470 MHz base clock and a 1650 MHz boost clock for the RTX 2060 Super, while our configuration uses a 1905 MHz clock. Also note that we follow Accel-Sim's representation of the memory clock as the *command* clock, which is twice as high as the commonly stated *data* clock.

In terms of resources, we reduce the number of SMs, the core clock, and the memory clock. For each resource we simulate 7 steps, meaning that we simulate the following ranges respectively:

SMs: 10, 15, 20, 25, 30, 35, 40.

Core clock: 1000, 1150, 1300, 1450, 1600, 1750, 1900.

Memory clock: 800, 1250, 1700, 2150, 2600, 3050, 3500.

We chose these ranges to represent realistic resource amounts[2], but note that the exact numbers are arbitrary; waste is not bound to specific numbers of resources.

[1]`https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-super.c3441`

[2]Also note that we increase the baseline (maximum) system to create increments that are easier to work with.

We focus on the these resources because they are (i) assumed to have the biggest impact on performance, (ii) easy to understand and/or reason about, (iii) applicable to all GPUs, and (iv) (excluding SMs) available to be altered dynamically. Future work could experiment with finer-grained resources such as functional units or caches. Note that some resource-types are system-wide, meaning that these resources cannot be shared. For example, if a core clock of 1000 MHz is optimal for kernel a, and 2000 MHz for kernel b, the best system for both (when co-locating or partitioning) is $\max(1000, 2000) = 2000$ MHz, not $1000 + 2000 = 3000$ MHz, or 1000 MHz for one portion and 2000 MHz for the other.

Because the time-to-complete per simulation varies between 8 and 96 hours, we cannot simulate the full Cartesian product of all these resources. Therefore, for each kernel, we only modify one resource at a time.

5.2 Applications

To demonstrate waste, we simulate four GPU-only applications from the Rodinia [1] benchmark: Hotspot, K-means, Backpropagation, and Breadth first search (BFS). For each kernel we reduce resources from the baseline system, simulate the execution, and analyze the results.

Application	Kernel	Workload size	Invocations	AI	Compute- / Memory-bound	Execution time share (baseline)
Hotspot	1	335 MB	1	4.16, 1.61[1]	M, C[2]	100%
K-means	1	426 MB	1	0.5[3]	M	46%
K-means	2	426 MB	2	125[4]	C	54%
Backpropagation	1	144 MB	1	0.22	M	17%
Backpropagation	2	144 MB	1	0.52	M	83%
BFS	1	1234 MB	13	0.06[5]	M	95%
BFS	2	1234 MB	13	0.17	M	5%

Table 5.1: Kernel-level characteristics for our selection of applications.

[1]For single-precision and double-precision respectively.

[2]For single-precision and double-precision respectively. If the compiler did not up the precision level, it would be only compute-bound.

[3]Integer operations / memory traffic.

[4]Flops / L1 (including shared and texture memory) traffic.

[5]Assuming an ‘average’ node at an ‘average’ level in the graph.

To show how our workflow can be used in practice we explain the kernels and their characteristics. Note that we assume the applications and their underlying algorithms are fully optimized, even though they are most likely not. The only thing we adjust are workload and parameters.

For each kernel we increased the default workload so it has a (full-system) execution-time of roughly 10 seconds or until the tracer ran out of memory. While analyzing the execution on a real GPU we found that copying data from and to the GPU takes up most of the execution time, with individual kernels running only for a fraction of the 10s goal. However, we did not change the workloads to increase this amount because it would increase the absolute differences between kernels (as the workload could not be increased for some applications), thereby making them harder to (visually) compare.

Table 5.1 shows the defining characteristics for our selected applications (listed below).

Hotspot Hotspot is a thermal simulation that iteratively solves a series of differential equations. The implementation features one kernel, named `calculate_temp`, that contains extensive indexing and coordination logic before it calculates the temperature in a loop surrounded by synchronization barriers. We use a 4096×4096 grid and 2 iterations. Because our number of iterations is small, the kernel is dominated by integer operations. Furthermore, although not indicative from the source, some the floating point instructions are actually executed with double-precision. Because this happens in the main computation portion of the code, double-precision instructions (`DADD` & `DFMA`) make up 35% of all floating-point instructions. Furthermore, Nsight Compute reports that the double-precision pipe is active 88% of the time.

We execute the application as follows:

```
./hotspot 4096 2 2 data/temp_4096 data/power_4096 output
```

Because hotpot is (largely) compute-bound, we expect to find waste in the memory hierarchy.

K-means K-means is a clustering algorithm. The implementation consists of two kernels. The first kernel, named `invert_mapping`, is a matrix transpose operation that copies data to and from main memory. The only computation involved is the loop and indexing (integer) logic. The second kernel, `kmeansPoint`, creates a fixed amount of clusters to which all input points are assigned. Points are assigned to clusters based on the closest distance of their features. Clusters are chosen based on the smallest root mean squared error (RMSE) to their respective points.

We use default parameters where applicable, e.g., the number of features, number of clusters, and number of loops are 12, 5, and 1 respectively. We adjust the default to workload to 3276800 elements to increase the execution time.

We execute the application as follows:

```
./kmeans -i data/3276800.txt
```

Because the kernels are memory- and compute-bound respectively, we expect to find compute-related waste for the `invert_mapping` kernel and memory-related waste for `kmeansPoint` kernel.

Backpropagation Backpropagation (or backprop) is a technique used in machine learning to update weights in a neural network based on the inputs. The implementation consists of two kernels: `bpnn_layerforward_CUDA`, which calculates the outputs based on the inputs and the weights of each neuron, and `bpnn_adjust_weights_cuda`, which updates the weights according to the resulting error rates from the other kernel. Both kernels primarily use single-precision floating point instructions. `bpnn_adjust_weights_cuda` also makes use of shared memory to store the weights, and relatively many barriers (5 out of 23 lines contain a `__syncthreads` call, including one in a loop) to ensure that the inputs travel layer-by-layer.

The workload can be defined by providing an amount of elements for the input layer. We set this value 1000000 and leave all other parameters to their default values, e.g., 16 hidden neurons and 1 output neuron.

We execute the application as follows:

```
./backprop 1000000
```

Because both kernels are memory-bound, we expect to find waste for compute-related resources.

Breadth first search (BFS) BFS is graph traversal algorithm that will for all current nodes first evaluate their neighbours before traversing any deeper in the graph. The implementation consists of two kernels, `Kernel1` and `Kernel2`. `Kernel1` will update the weights of all the neighbours for all the nodes of the current level. `Kernel2`, which will select the nodes to traverse in the next level (and check if all nodes have been traversed), is always directly executed after `Kernel1`. The simulation finishes once all nodes have been visited.

We use a workload of 16M nodes with, on average, 8 randomly connected edges. Depending on the graph, the current iteration might visit many more or many less nodes than previous iterations. This means that BFS, unlike other kernels, will execute different numbers of instructions per kernel invocation.

We execute the application as follows:

```
./bfs data/graph16M.txt
```

Both BFS kernels only make use of integer and memory operations. Because both kernels are memory-bound, we expect to see waste for compute-related resources.

6

Evaluation

In this section we demonstrate the analysis part of our workflow for finding, measuring, and combining waste. We first look at each modified resource and identify the waste for each kernel. Following that, we reevaluate our findings with a looser definition of waste, attempt to combine resources, and evaluate our optimal systems for the other kernels. Lastly, we analyze and discuss potential waste markers.

6.1 Observed waste

We observe waste when a system with fewer resources than the baseline yields equal or better results for either execution time or energy. Note that the choice of, and the individual importance of, the performance metrics is ultimately up to the user.

	Execution time (ms)							Energy (J)						
SMs	10	15	20	25	30	35	40	10	15	20	25	30	35	40
hotspot	13.4	9.0	6.7	5.4	4.5	3.8	**3.4**	1.3	0.9	0.7	0.6	0.5	0.4	**0.4**
k-means (1)	5.5	3.9	3.2	**2.7**	2.9	5.7	7.4	0.6	0.5	0.4	**0.4**	0.4	0.7	0.9
k-means (2)	13.0	8.7	6.5	5.2	4.4	3.8	**3.3**	1.2	0.8	0.7	0.5	0.5	0.4	**0.4**
backpropagation (1)	3.4	2.3	1.7	1.4	1.1	1.0	**0.9**	0.4	0.3	0.2	0.2	0.1	0.1	**0.1**
backpropagation (2)	1.4	0.9	0.7	0.6	0.5	0.4	**0.4**	0.2	0.1	0.1	0.1	0.1	0.1	**0.1**
BFS (1)	49.3	40.2	36.8	35.2	34.3	33.8	**33.5**	4.5	3.8	3.6	3.5	**3.5**	3.5	3.6
BFS (2)	6.1	4.1	3.1	2.8	2.5	2.4	**2.2**	0.6	0.4	0.3	0.3	0.3	0.3	**0.3**

Table 6.1: Performance in time and energy for various number of SMs. The baseline has 40 SMs. Bold values denote the optimal resource amount.

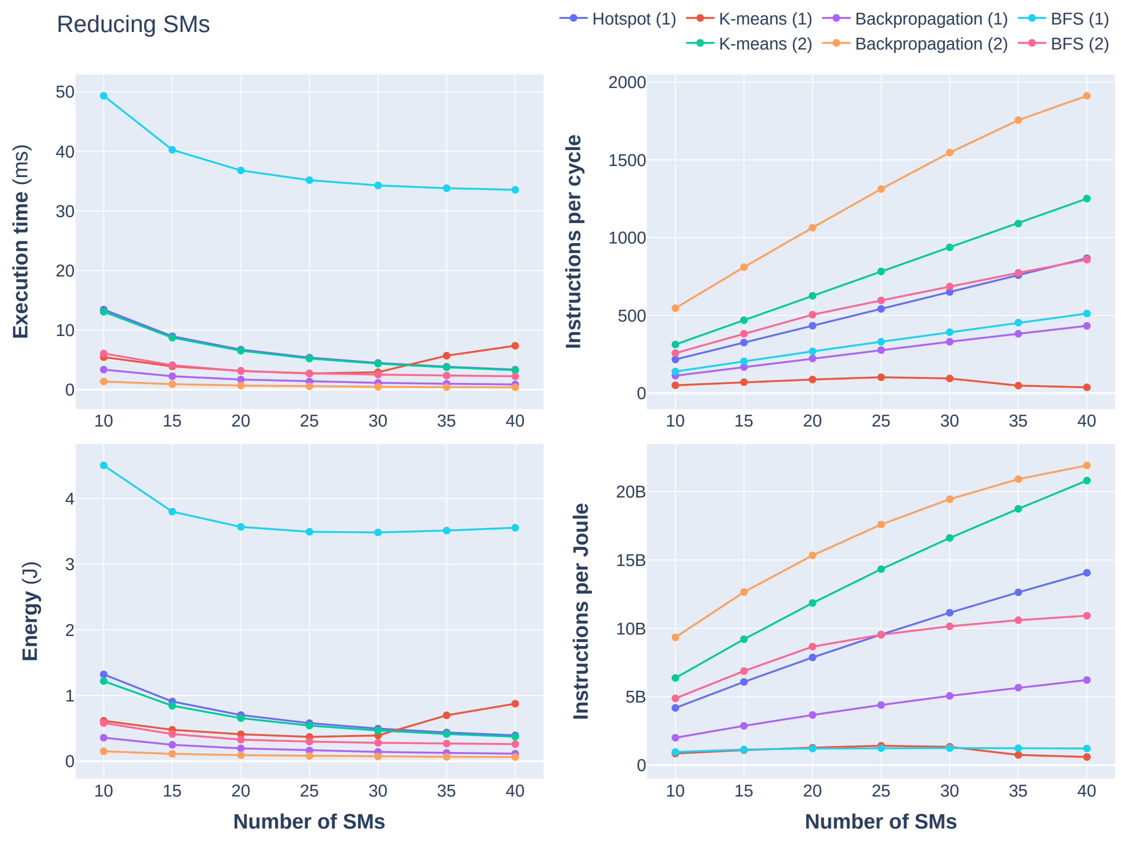

Figure 6.1: Performance and efficiency for various number of SMs. The baseline has 40 SMs.

6.1.1 SMs

SMs are the highest-level (compute) building block of the GPU. Because a portion of work is (mostly) confined to single SM both in software (as thread blocks) and hardware (as warps), the SM is an inherently scaleable resource. Each SM contains a selection of different (e.g., integer, double-precision floating point) cores and a L1 cache, including shared and texture memory. Only the L2, the main memory, and the interconnect lie outside of the SMs. Increasing the number of SMs should allow the compute to be spread out among more cores, meaning that more warps can be executed in the same cycle. Likewise, decreasing the amount of SM's will decrease the total amount of possible warps per cycle.

When increasing the number of SMs, the advantages are:

+ more (available) simultaneous compute (and thus more compute per second),
+ more potential to share L2 lines,
+ a higher (combined) L1 size, and
+ a higher (combined) L1 throughput.

Conversely, potential drawbacks are:

– fewer warps to choose from (per SM), and as such, less potential to hide memory latencies,
– more contention for the L2 cache (and, potentially, if the L2 cache becomes overwhelmed, more contention for the main memory), and
– increased power consumption.

SM waste can occur if the memory becomes overwhelmed due to extra memory traffic or because the kernel cannot make use of the extra compute, in which case the extra power can outweigh any potential gains.

The results can be seen in Figure 6.1 and Table 6.1. First of, we see that all kernels except **k-means (1)** achieve a lower execution time when increasing the number of SMs (on average: with $\sim 75\%$ from 10 to 40 SMs). As such, we cannot find waste for these kernels; every extra SM leads to an increase in execution time.

K-means (1) is the only kernel where decreasing the number of SMs not only yields the same performance but rather significantly better performance. Executing the kernel on a

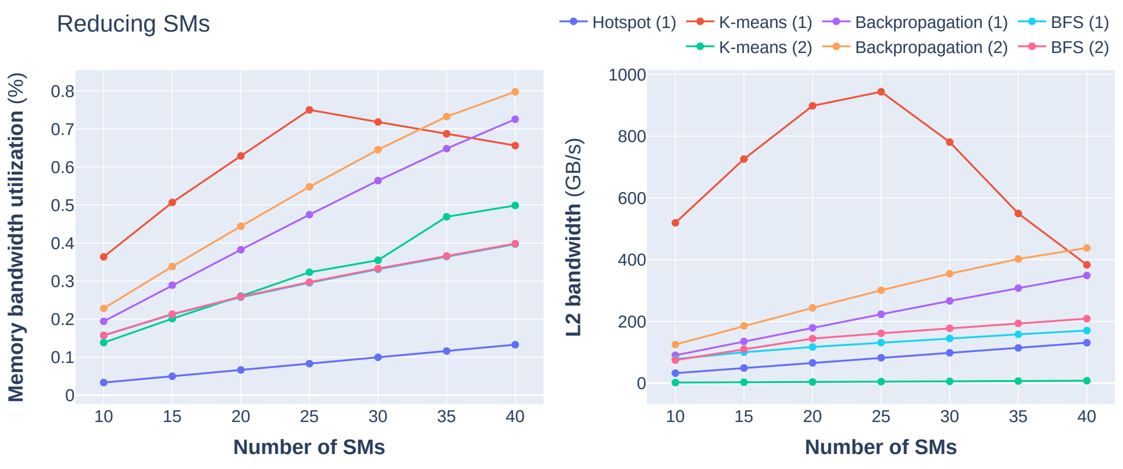

Figure 6.2: Memory and L2 bandwidth utilization for various number of SMs.

system with 25 SMs instead of baseline 40 takes 63% less time while consuming 58% less energy. Although the reason for the performance increase must be related to the memory, as the compute power scales with the number of SMs, we are unclear of the exact cause. The average memory latency spikes from $\sim$ 60 cycles at 25 SMs to $\sim$ 500 cycles at 40 SMs. In comparison, the other kernels average around 40 cycles for all number of SMs. The obvious explanation is more contention for either L2 or the main memory, causing higher latencies and lower throughput (relative to the number of threads). However, the simulator reports that the memory and L2 bandwidth utilization actually decreases from 25 SMs onwards, as can be seen in Figure 6.2.

We also observe that BFS (1) consumes 2% less energy at 30 SMs, even though it takes 2% longer. Because the optimal resource amounts conflict between performance metrics, we do not classify this as waste.

Overall, k-means (1) proves that adding *more* resources can *lower* the performance. In this case, using more SMs is not only wasteful, it is also harmful.

In terms of efficiency, the compute-bound kernels gain a steady amount of IPC per SM increase: $\sim$ 310 per 5 SMs for k-means (2) and $\sim$ 110 for hotspot. We therefore expect these kernels to continue to benefit from more SMs beyond the baseline. The increase in IPC for Backpropagation (1) and backpropagation (2) however decelerates at higher numbers of SMs, e.g., backpropagation (2) gains $\sim$ 263 IPC from 10 to 15 SMs but only $\sim$ 156 from 35 to 40 SMs. The energy efficiency displays a similar trend. We therefore expect the optimal number of SMs to be lower than for the compute-bound kernels (but

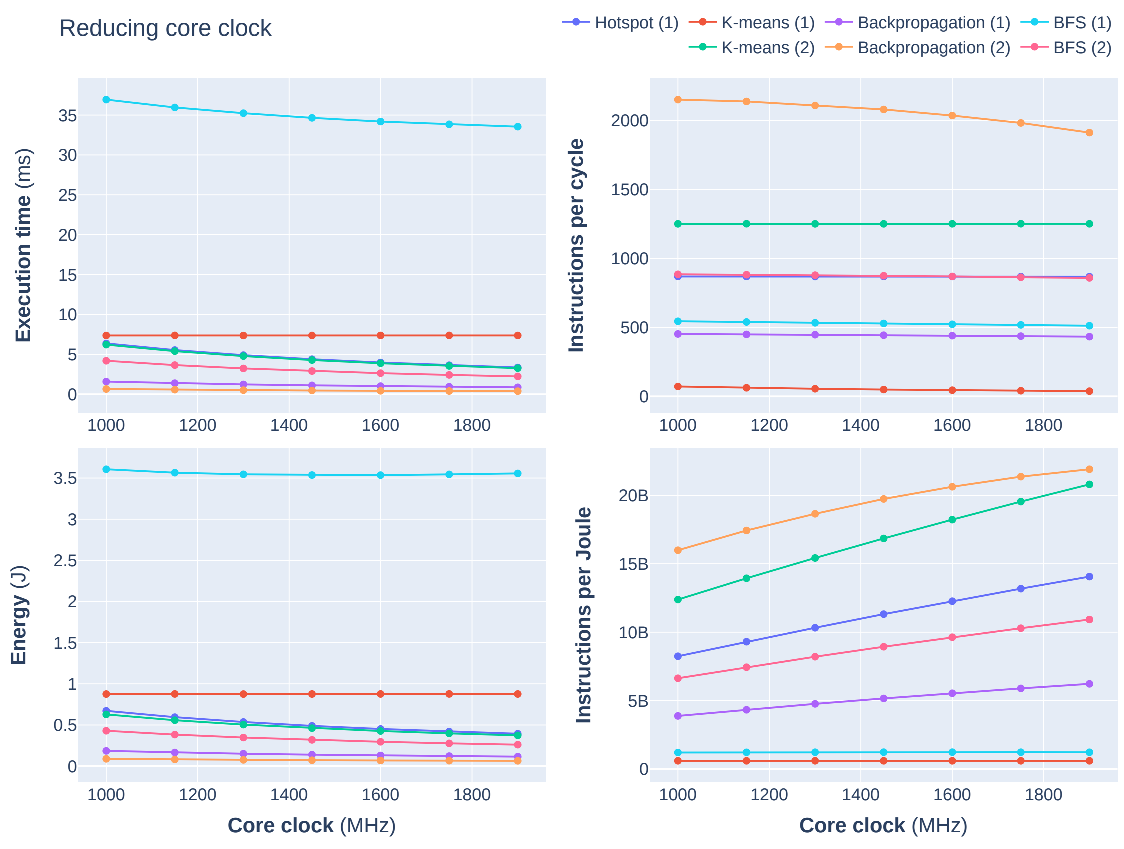

Figure 6.3: Performance and efficiency for various core clocks. The baseline has a core clock of 1900 MHz.

still beyond the baseline).

One could also view the the absolute difference between the peak and achieved IPC or instructions per Joule as waste. For instance, hotspot can issue 1643 fewer instructions at 40 SMs than k-means (2). Instead of the system wasting extra SMs, like with k-means (1), the system continually wastes a portion of each individual SM. We currently do not view this absolute difference as waste because it not correlated with performance. Furthermore, this ‘waste’ is likely fixed. An application with 10× more memory instructions than compute instructions will never utilize the full SM(s), and can therefore never reach the same IPC as an application more equally matched to the hardware.

6.1.2 Core clock

The core clock (frequency) defines the rate of cycles per second each core can execute. A higher frequency, i.e., more cycles per second, allows for more instructions to be issued in

	Execution time (ms)							Energy (J)						
Core clock	1000	1150	1300	1450	1600	1750	1900	1000	1150	1300	1450	1600	1750	1900
hotspot	6.4	5.5	4.9	4.4	4.0	3.6	**3.4**	0.7	0.6	0.5	0.5	0.5	0.4	**0.4**
k-means (1)	**7.4**	7.4	7.4	7.4	7.4	7.4	7.4	**0.9**	0.9	0.9	0.9	0.9	0.9	0.9
k-means (2)	6.2	5.4	4.8	4.3	3.9	3.6	**3.3**	0.6	0.6	0.5	0.5	0.4	0.4	**0.4**
backpropagation (1)	1.6	1.4	1.2	1.1	1.0	0.9	**0.9**	0.2	0.2	0.2	0.1	0.1	0.1	**0.1**
backpropagation (2)	0.7	0.6	0.5	0.5	0.4	0.4	**0.4**	0.1	0.1	0.1	0.1	0.1	0.1	**0.1**
BFS (1)	36.9	36.0	35.2	34.6	34.2	33.8	**33.5**	3.6	3.6	3.5	3.5	**3.5**	3.5	3.6
BFS (2)	4.2	3.7	3.2	2.9	2.6	2.4	**2.2**	0.4	0.4	0.3	0.3	0.3	0.3	**0.3**

Table 6.2: Performance in time and energy for various core clocks (MHz). The baseline has a 1900 MHz core clock. Bold values denote the optimal resource amount.

the same time span. Unlike reducing SMs, reducing the core clock does not change the spread, so we expect the amount of contention and synchronization to stay the same across frequencies.

When increasing the core clock frequency, possible advantages are:

+ more compute per second, and
+ a higher l2 throughput (as the L2 clock is set to the core clock).

The only drawback is:

− increased power consumption.

Core clock waste can occur if (i) the extra power consumption is far greater than the extra compute per second or L2 throughput, or (ii) if the extra compute per second or L2 throughput cannot be fully utilized. (ii) can occur if there is not enough compute instructions to hide the memory latencies. Because the main memory runs on a different clock frequency, memory loads, provided that they cannot be fetched from L1 or L2, take the same amount of time regardless of the core clock. In this case, increasing the core clock will require more compute instructions to be issued within the same time span to achieve the same SM utilization. If this is is not possible, it can be beneficial to lower the core clock in order to save power.

For these reasons, we expect to find core clock waste in applications with more (main) memory operations than compute operations, i.e., the memory-bound kernels. Of our selection, **k-means (1)**, **backpropagation (1)**, **backpropagation (2)**, **BFS (1)**, and

BFS (2) can be considered memory-bound. The results of reducing the core clock can be seen in Figure 6.3 and Table 6.2.

We observe that k-means (1) is unaffected by the core clock, both in terms of time and energy. Because the performance remains equal, we classify the unused extra core clock (from 1000 to 1900 MHz) as waste. We suspect the cause of this waste due to reason (ii) because k-means (1) mostly consists of memory instructions, and because (i) is not possible as the energy does not increase. Unlike the SMs, the optimal core clock seems to be even lower than our lowest simulated clock. We expect that the core clock can be further reduced until the cycles per second becomes lower than the the amount of possible completed memory operations per second (which is partly based on the memory clock). The exact clock however depends on scheduling and cache policies and is therefore difficult if not impossible to predict.

From the time and energy figures of k-means (1), the results seem to indicate that the core clock has no effect on the power consumption. The direct power consumption results in Figure 6.7 seem to support this observation. We suspect this is because the Accel-sim [3] power model, AccelWattch [47], does not include the core clock for its calculation of the static power. The power consumption of the clocks is only taken into account in the dynamic power, i.e. the power per (relevant) executed operation. Because the core clock has little influence on the dynamic power (as k-means (1) is dominated by memory, not compute, operations), the total power remains unaffected.

We belief this behaviour is incorrect. Intuitively, the power consumption should decrease when reducing the core clock. We tested the effect of the core and memory clock on the power consumption on a real GPU[1]. To determine if this static power trend is incorrect, we use k-means (1) and hotspot because, in the simulations, their power consumption does not change when reducing the core and memory clock respectively (meaning that the total power is equal to the static power). The results can be seen in Figure 6.4. Note that we increase the total amount of invocations to get a reliable power measurement. From the measurements we can see that the power increases with both the core and memory clock, with 50% and 300% respectively. Although the exact increase is likely different across systems, we believe this general behaviour is universal. Previous work seems to support our findings [49]. Thus, our findings imply that our previous observations on waste are merely an upper bound. In reality we would expect to use less power with a lower clock and, with the observed execution times, also less energy.

[1]Measured on a Nvidia GeForce RTX 3050 Ti Laptop GPU using nvidia-smi. Frequencies were chosen to resemble the simulated configurations whenever possible.

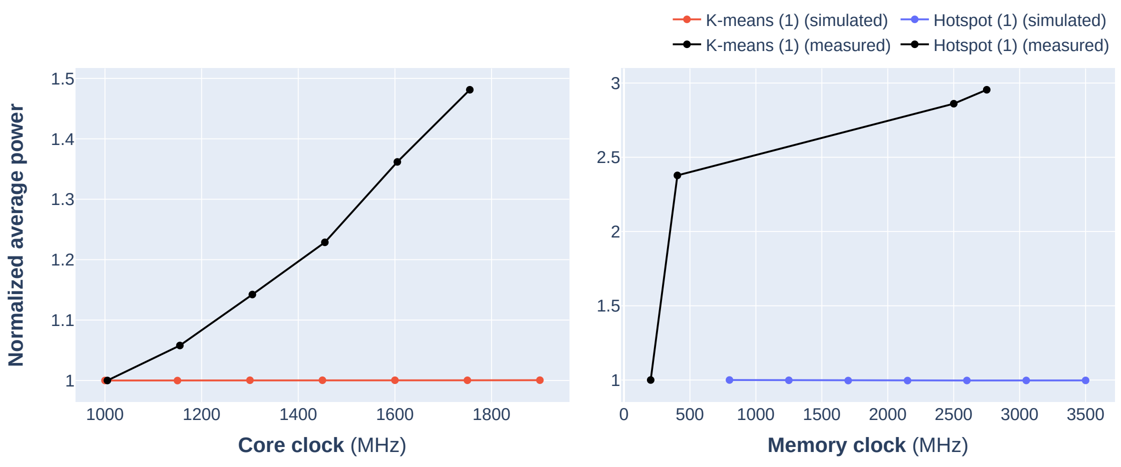

Figure 6.4: Measured versus simulated power consumption for various core and memory clocks. Normalized to the power of the lowest clock.

Similar to the SM-waste, BFS (1)'s optimal energy is achieved at a lower core clock (1600 MHz) than the optimal clock for the time, as shown in Table 6.2. Again, because the performance metrics are inconclusive, we do not consider this waste.

The other kernels, including the remaining memory-bound kernels, backpropagation (1), backpropagation (2), and BFS (2), do not exhibit waste. The increase in performance between the lowest and the highest core clock is similar, between $41 - 47\%$ for the execution time and $27 - 41\%$ for energy. It is unclear why the memory-bound kernels are roughly equally affected by the core clock as the the compute-bound kernels. It is possible this is simply due to the extra L2 throughput.

We see a decrease in efficiency when reducing the core clock for backpropagation (2) but an increase in energy efficiency with the same kernel. We assume this is because the instructions per Joule incorporates the core clock while the IPC does not.

6.1.3 Memory clock

The memory clock (frequency) determines the rate at which the (on-chip) main memory can handle requests. Unlike components such as the bus width or the memory controllers, varying the memory clock will change the latency (and therefore throughput), not the bandwidth. Therefore, when increasing the memory clock we expect to see:

+ a lower memory latency, and

+ a higher memory throughput.

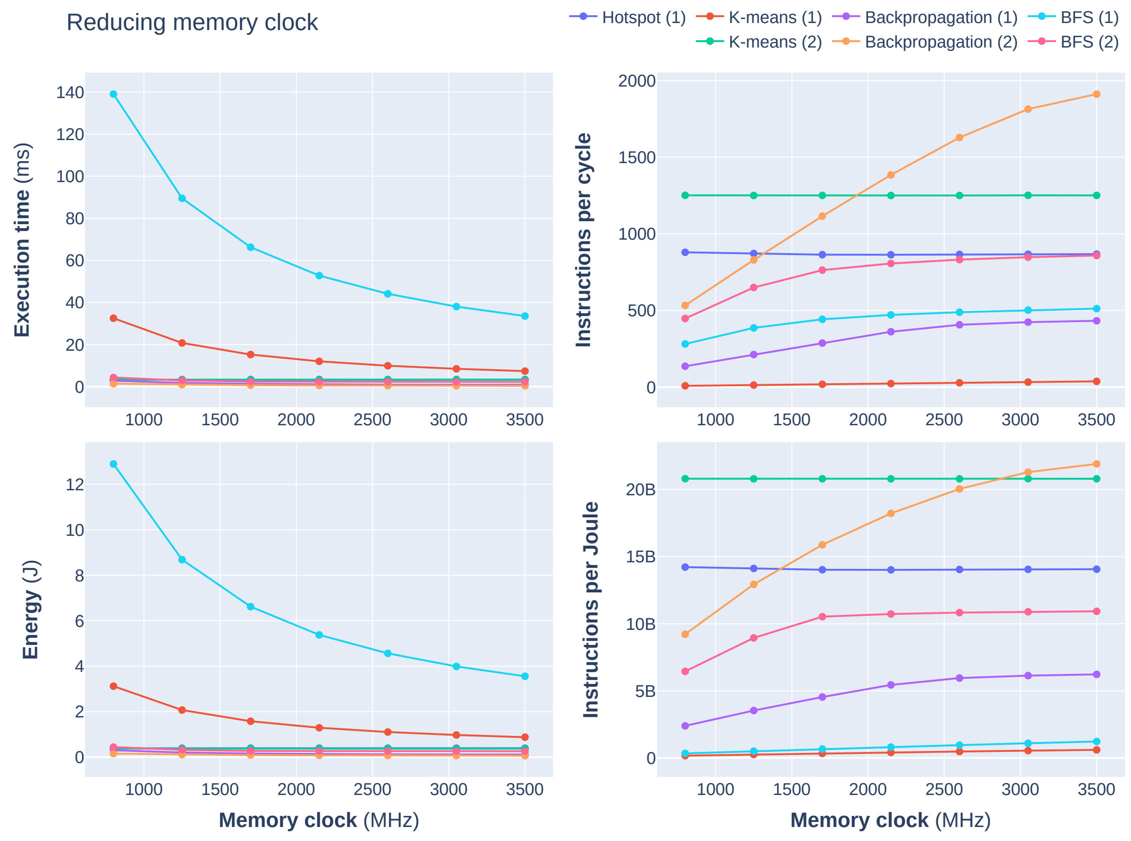

Figure 6.5: Performance and efficiency for various memory clocks. The baseline has a memory clock of 3500 MHz.

	Execution time (ms)							Energy (J)						
Memory clock	800	1250	1700	2150	2600	3050	3500	800	1250	1700	2150	2600	3050	3500
hotspot	**3.3**	3.3	3.4	3.4	3.4	3.4	3.4	**0.4**	0.4	0.4	0.4	0.4	0.4	0.4
k-means (1)	32.5	20.8	15.2	12.0	9.9	8.5	**7.4**	3.1	2.0	1.6	1.3	1.1	1.0	**0.9**
k-means (2)	**3.3**	3.3	3.3	3.3	3.3	3.3	3.3	**0.4**	0.4	0.4	0.4	0.4	0.4	0.4
backpropagation (1)	2.8	1.8	1.3	1.0	0.9	0.9	**0.9**	0.3	0.2	0.2	0.1	0.1	0.1	**0.1**
backpropagation (2)	1.4	0.9	0.7	0.5	0.5	0.4	**0.4**	0.2	0.1	0.1	0.1	0.1	0.1	**0.1**
BFS (1)	139.0	89.4	66.2	52.8	44.1	38.0	**33.5**	12.9	8.7	6.6	5.4	4.6	4.0	**3.6**
BFS (2)	4.4	2.9	2.4	2.3	2.3	2.3	**2.2**	0.4	0.3	0.3	0.3	0.3	0.3	**0.3**

Table 6.3: Performance in time and energy for various memory clocks (MHz). The baseline has a 3500 MHz memory clock. Bold values denote the optimal resource amount.

With:

– increased power consumption.

We expect the memory-bound kernels to benefit the most from a higher memory clock, so long as the time savings outweigh the extra power. Conversely, we expect to find waste within the compute-bound kernels: **hotspot** and **k-means (2)**. These kernels cannot benefit from the lower memory latency or the higher throughput, simply because the amount of compute is already dominating. Increasing the memory clock will only exacerbate this balance. Note that the memory clock only affects the main memory. Any kernel, compute- or memory-bound, that primarily deals with L2 or lower might not be affected (thereby creating waste).

The results of reducing the memory clock can be seen in Figure 6.5 and Table 6.3. Indeed, we find that both compute-bound kernels do not drop in performance when lowering the memory clock. As such, we view the extra memory clock (from 800 to 3500 MHz) as waste for these kernels. All other kernels lose both time and energy for any memory clock lower than the baseline.

Like the core clock, the memory clock does not seem to affect (static) power in the simulations. As demonstrated in subsection 6.1.2, this behaviour is most likely incorrect. Depending on the intensity of the real power increase, waste could potentially also exist in the **backpropagation (1)**, **backpropagation (2)**, **BFS (1)** and **BFS (2)** kernels because the decrease in energy slows down for higher memory clocks, i.e., the efficiency flattens, as can also be seen in the instructions per cycle/Joule graphs.

6.1.4 Efficiency

Heretofore, we have looked at waste as reducing resources without compromising performance. We use this definition to show how waste is always bad, i.e., there is no counterargument to advocate for keeping underutilized resources that do not benefit performance in any way. However, in practice, one might want to trade resources for performance, especially if the the ratio of extra resources to performance, i.e. the efficiency, stagnates. For instance, the IPC and instructions per Joule in Figure 6.1, Figure 6.3, and Figure 6.5 for **backpropagation (1)** and **backpropagation (2)** repeatedly levels off or dives at higher levels of resources.

Table 6.4 demonstrates our workflow with a different definition of waste that allows for either a 10% or 20% drop in performance compared to the 0% baseline. We observe

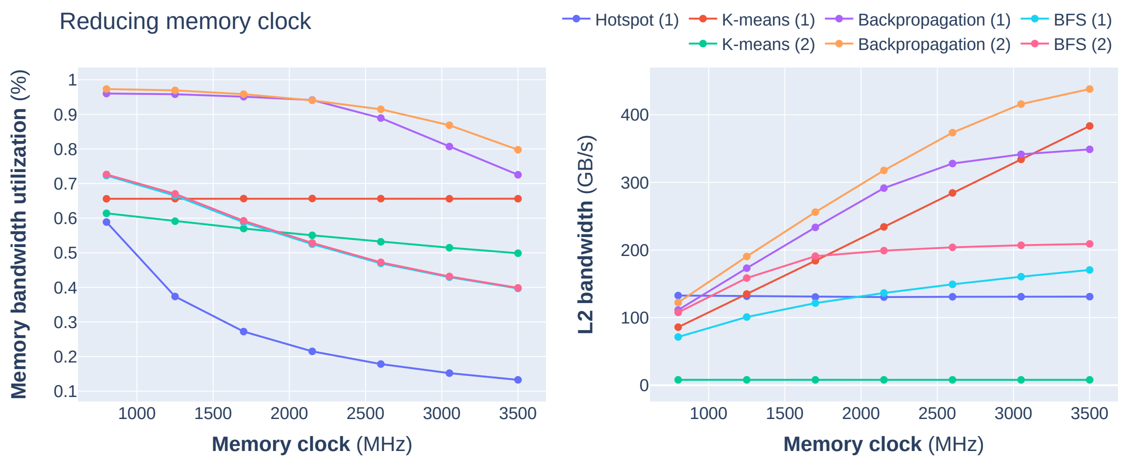

Figure 6.6: Memory and L2 bandwidth utilization for various memory clocks.

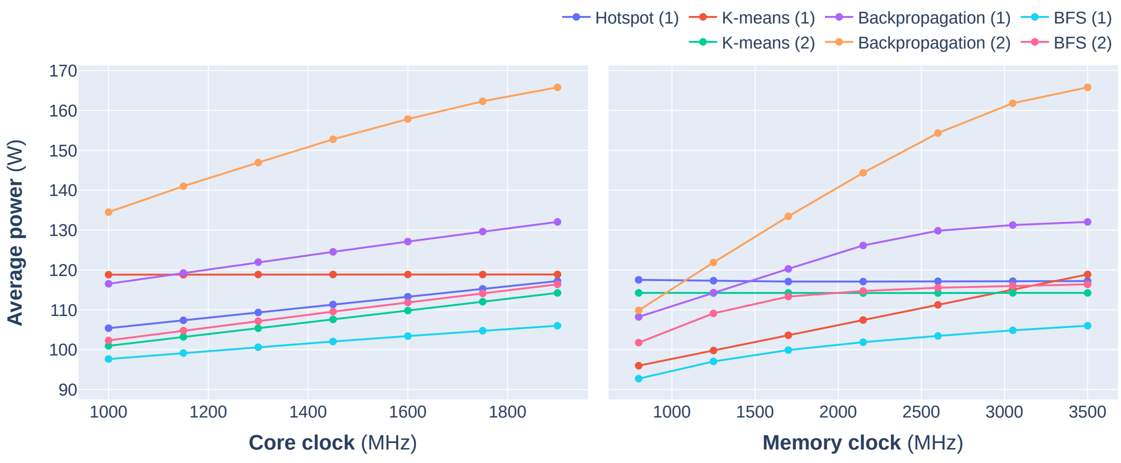

Figure 6.7: Power consumption for various core, and memory clocks.

that for kernels, except **BFS (1)** and **BFS (2)**, the results are not significantly different. The largest reduction of resources occurs for the **BFS (1)** kernel which can reduce up to $\sim 50\%$ of the SMs and the core clock in return for a 10% reduction in performance. As can be observed from the efficiency graphs, the amount of resources that can be reduced will decrease with an increase in the tolerated performance penalty. In other words, tolerating a 5% performance penalty will reduce relatively more resources than a 50% penalty.

Like with the (potential) SM-waste for **BFS (1)** (see subsection 6.1.1), the results for 10% and 20% present situations where the optimal system is different between performance metrics, specifically: situations where the number of resources has a smaller impact on energy than on time. For example, reducing the SMs for **backpropagation (1)** to 35 will yield 9.84% more energy but 13.27% more time. Similarly, reducing the memory clock for **backpropagation (2)** to 2600 will result in 9.28% more energy but 17.40% more time. Even though the actual difference is likely smaller because of inaccuracies in the power model (see subsection 6.1.2), these situations can still occur, and must be dealt with. For example, one could define the importance of each metric or use a relative or absolute maximum for each (or all) metrics

6.2 Effect of workload

As stated in our definition of waste (2), in order to find system waste we must fix the application and workload. In this section however, we vary the workload, and analyze the results to see if the same waste patterns remain. Specifically, we vary the workload *size*.

To demonstrate the effect of the workload size, we re-simulate **k-means (1)** and **k-means (2)** with the total workload size halved and doubled (to 1638400 and 6553600 elements respectively). We use K-means because it has both a memory-bound and a compute-bound kernel, which might yield different results. Figure 6.8 shows the performance results for each kernel for different workload sizes and for increasing number of SMs. We observe that each workload size displays similar trends across both kernels. Furthermore, the efficiency and waste is almost exactly equal compared to the default workload size.

On the memory-side, we suspect that this is true because if the SM-L2 or L2-memory bus is 100% saturated with the original workload, doubling the workload will force the extra requests to be sequential, meaning that our IPC remains the same, and only the the total number of cycles changes (in fact, exactly doubles).

		Tolerated performance penalty				
		Both	Time		Energy	
Kernel	Resource	0%	10%	20%	10%	20%
hotspot	SM	40	40	**35**	40	**35**
	Core clock	1900	**1750**	**1600**	**1750**	**1600**
	Memory clock	800	800	800	800	800
k-means (1)	SM	25	25	**20**	25	**20**
	Core clock	1000	1000	1000	1000	1000
	Memory clock	3500	3500	**3050**	3500	**3050**
k-means (2)	SM	40	40	**35**	40	**35**
	Core clock	1900	**1750**	**1600**	**1750**	**1600**
	Memory clock	800	800	800	800	800
backpropagation (1)	SM	40	40	**35**	**35**	**35**
	Core clock	1900	**1750**	**1600**	**1750**	**1600**
	Memory clock	3500	**2600**	**2150**	**2600**	**2150**
backpropagation (2)	SM	40	**35**	**35**	**35**	**30**
	Core clock	1900	**1750**	**1600**	**1750**	**1600**
	Memory clock	3500	**3050**	**2600**	**2600**	**2600**
BFS (1)	SM	40	**20**	**20**	**15**	**15**
	Core clock	1900	**1150**	**1000**	**1000**	**1000**
	Memory clock	3500	3500	**3050**	3500	**3050**
BFS (2)	SM	40	**35**	**30**	**30**	**25**
	Core clock	1900	**1750**	**1600**	**1750**	**1600**
	Memory clock	3500	**1700**	**1700**	**1700**	**1700**

Table 6.4: Optimal resource amounts for a relaxed definition of waste. Bold values are lower than than the 0% baseline. Lower is better.

On the compute-side, the workload size likely does not alter the composition nor the order of instructions, meaning that at any time during the execution, its behaviour will be identical. Increasing the workload size will either cause a loop in the kernel to have more (or less) iterations or cause the kernels to execute more (or less) blocks. Both are sequential additions or reductions. Furthermore, increasing the number of blocks, will increase the amount of warps the warp scheduler can choose from, which could therefore, in theory, hide more latencies. However, because adjusting the workload size does not alter the instruction mix, the warp scheduler must still pick from the same selection of warps. We suspect the only difference is that there will be more warps waiting in the queue.

However, we suspect this will only happen if the if the SMs are fully 'filled'. If the workload is not large enough to saturate the SMs, the amount of active warps will be lower (than the peak at larger workload sizes), meaning that there could be less contention for resources, e.g., fewer filled cache lines or less main memory traffic. In this situation, increasing the workload size could actually harm performance.

We believe the observations on the size of the workload will hold for other applications. However, we cannot make claims on other aspects of the workload, such as the density, fidelity, or other domain-specific characteristics, as unlike the size, they may change the instruction mix.

6.3 Optimal system

In previous sections, we observed the following three outcomes:

1. −Waste, =performance: reducing the memory clock in the optimal system for **hotspot** and **k-means (2)** results in a negligible decrease in time and energy. Similarly, reducing the core clock for **k-means (1)** does not impact performance. This is expected behaviour, as unlike SMs, the core and memory clock beyond the optimal point[1] cannot hurt nor aid performance because these kernels are dominated by the opposite type of operations, i.e., memory or compute respectively. An import caveat is that the power, and with a static time, also energy, will increase with more resources.

[1]Note that the same holds for kernels that did not reach this point in our selection of clocks. For example, there exists a core clock where the (compute-bound) **hotspot** will be able to execute its (dominating number of) compute operations faster than its memory operations (assuming the number of main memory requests is nonzero).

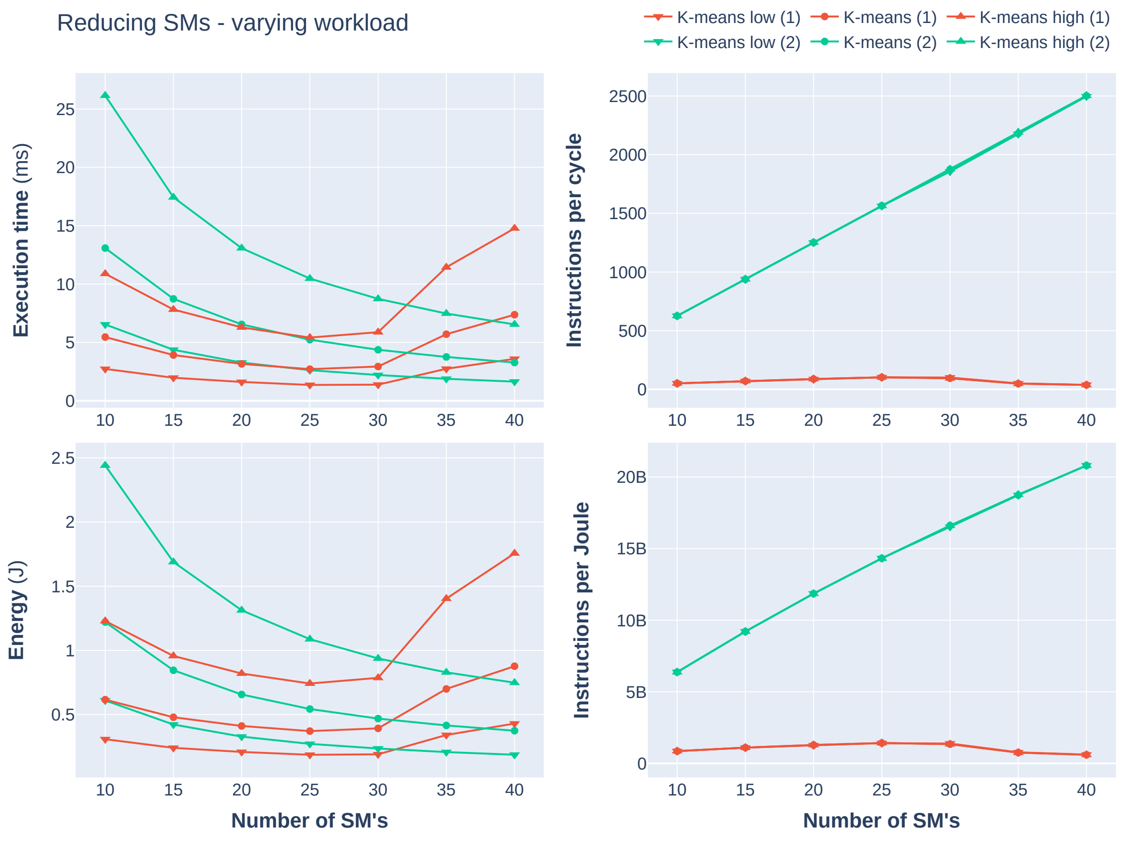

Figure 6.8: Performance and efficiency for different workload sizes, for various number of SMs.

2. −Waste, +performance: reducing the number of SMs for **k-means (1)** will result in a lower execution time and a lower energy consumption. The intuition that more resources will deliver equal or better performance is not true.

3. =Waste, =performance: The optimal system for **backpropagation (1)**, **backpropagation (2)**, **BFS (1)**, and **BFS (2)** is the baseline system.

At most, we were able to reduce $\sim 37\%$ of SMs, $\sim 47\%$ of the core clock, and $\sim 77\%$ of the memory clock. We suspect to be able to reduce even more of the clocks as the efficiencies show no sign of increasing at the lower bounds of the resource range(s).

In our workflow, we adjust each resource independently. This raises the question: can we combine the optimal levels we observed for each resource into the same (kernel-specific) system? Figure 6.9 depicts the time and energy for the baseline system, the optimal system for each individual resource, and the system that trivially combines the individual optimal systems. Note that the optimal system for **backpropagation (1)**, **backpropagation (2)**, **BFS (1)** and **BFS (2)** is the baseline system itself. **Hotspot** and **k-means (2)** only have one resource that deviates from the baseline. For these kernels, the optimal combined system is therefore the optimal system for resource that was reduced. Only **k-means (1)** harbours multiple sources of individual waste: the number of SMs (25 vs. 40) and the core clock (1000 vs. 1900). We observe that when combining both types of waste, the optimal combined system (with 25 SMs and a 1000 MHz core clock) performs worse than the optimal SM system, in both time and energy. Thus, it seems that we cannot trivially combine waste. The actually optimal combined system will most likely be somewhere between the optimal SM system and the optimal core clock system.

We can therefore only conclude that, for **k-means (1)**, the optimal SM system outperforms the baseline, the optimal core clock system, and the optimal combined system.

6.4 Optimal general-purpose system

Using the optimal systems as defined in section 6.3, we inspected how our optimal systems would perform with the other kernels. Figure 6.10 shows the relative increase or decrease in performance (to the baseline) of each kernel for each optimal system compared to the baseline. Note that we omit the kernels for which the baseline system is the optimal system. The first observation is that non of the systems outperform the baseline consistently. Only the optimal systems for **hotspot** and **k-means (2)** yield better results for both respective kernels. This is not surprising as their optimal systems are identical, i.e., the baseline

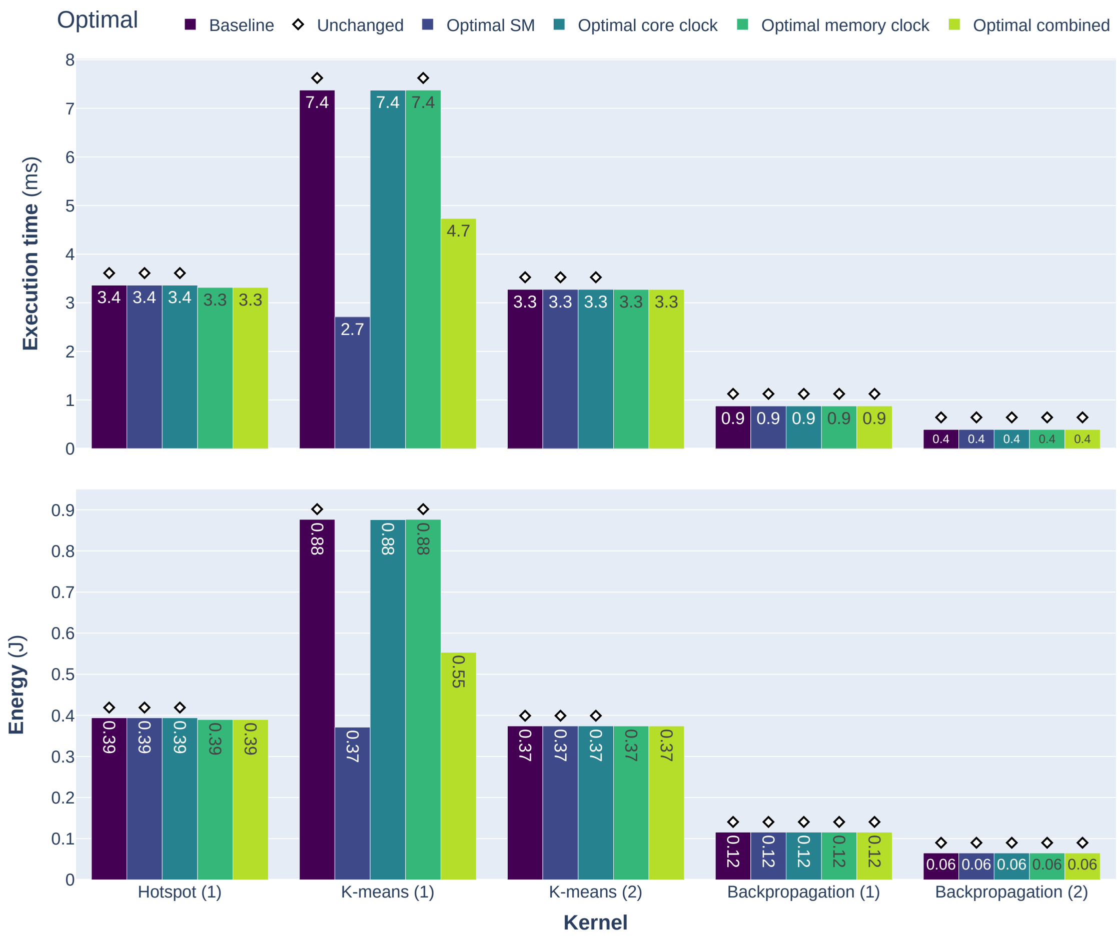

Figure 6.9: Performance for optimal systems. Lower is better.

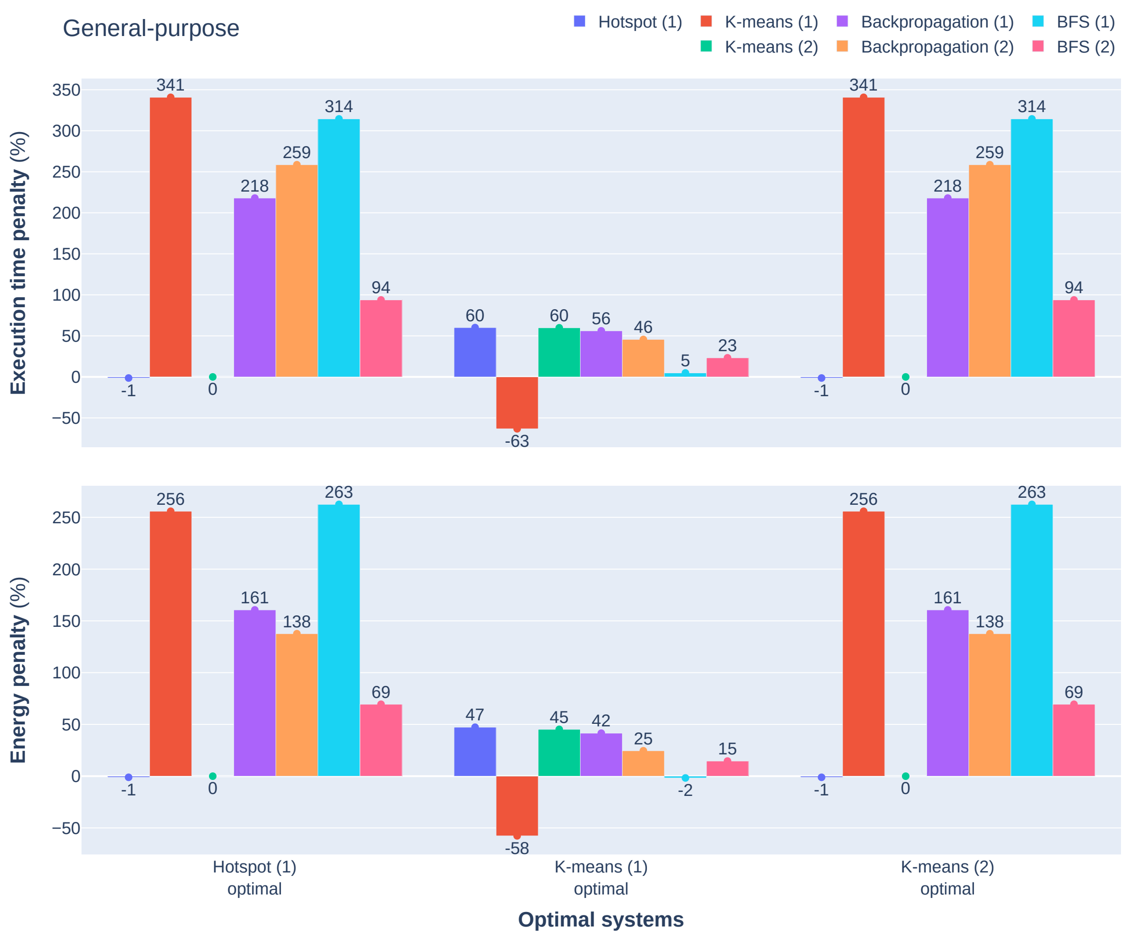

Figure 6.10: Performance penalty for optimal general-purpose systems (different from the baseline). Lower is better.

system with the lowest possible memory clock. Although this system reduces waste for their kernels, it drastically decreases performance for the others (up to 256%). Overall, we find that all (non-baseline) optimal systems incur a performance loss (up to 341%).

Therefore, for general-purpose computing, reducing SMs, the core clock, or the memory clock does not produce an overall better performing system.

6.5 Alternative indicators of waste

This section discusses possible waste indicators from within our (and Accel-Sim's) collection of metrics.

Utilization Utilization is inherently closely tied to waste. It can show if, and how much, a resource is used. However, using utilization as a sole indicator of waste brings complications such as:

- The utilization of many resources cannot be measured directly. For instance, one can only define the utilization of a SM (or SMs) as the utilization of finer-grained resources, e.g., the number of active single-precision compute units. Alternatively, one can use utilization metrics comprised of multiple resources, e.g. the memory bandwidth. Unfortunately, these metrics are difficult to trace back to the original resource, e.g., which resources should be reduced for the memory bandwidth?: the core clock (which control the L2 cache), the memory clock, the number of memory controllers, etc.? Both approaches could most likely not have predicted the SM-level waste for **k-means (1)** as shown in subsection 6.1.1.
- The number of utilization metrics is expected to be much larger than the number of performance metrics. As such, the complexity of combining and comparing metrics grows. As an example, for **backpropagation (1)** and **backpropagation (2)**, the best memory bandwidth utilization is achieved at 1000 MHz while the highest L2 bandwidth is achieved at 3500 MHz (see Figure 6.6).
- Improving utilization might harm performance. As shown in figure Figure 6.6, the highest memory bandwidth utilization is achieved (for all kernels) with a memory clock of 1000 MHz. However, Figure 6.5 shows that this is the worst performing clock for **k-means (1)**, **backpropagation (1)**, and **backpropagation (2)**.
- To ensure the configuration that can 'optimally' decrease underutilization did not harm performance, a performance model must be used to predict performance (as the hardware does not exist) and compare it to the baseline. And, as shown by our workflow, if a performance model must be used to predict performance, it can also be used to predict waste.

Efficiency Another approach is to analyze the efficiency, i.e., IPC and instructions per Joule, instead of the performance. Based on the results of the simulations, the highest efficiency almost always correlates with the best performance. The only exception is the core clock, simply because it defines the length of a cycle, something the IPC (instructions *per cycle*) cannot describe. Because instructions per Joule is based on the core clock (through the execution time), it does not face the same problem. We observe that selecting the highest instructions per Joule (at the lowest resource amount) yields the same optimal resource amounts as the performance metrics do. In other words, we find the same waste.

Currently, there is no method to predict these efficiency metrics outside the enveloping performance models. However, a light-weight efficiency prediction model could perhaps be a viable alternative in the future.

Occupancy Occupancy, the number of active warps compared to the theoretical maximum, is a type of utilization metric, but noteworthy because it can be calculated at compile-time. It is calculated using the number of threads per block, the number of registers per thread, and the amount of shared memory per block (along with several system characteristics). At run-time, occupancy is measured simply by counting the number of average active warps. Figure 6.2 depicts the theoretical and achieved occupancy for our kernels, varied in the number of SMs.

The first observation is that the theoretical occupancy is always 100%, for all SMs. The latter is expected because occupancy is simply a per-SM metric. The former is also not unexpected because a 100% theoretical occupancy only implies that the hardware is able to accommodate the maximum number of warps, not that the hardware is fully utilized (e.g., using the optimal number of registers per thread). It is therefore difficult to decide from a 100% theoretical occupancy if, which, and how many resources should be reduced.

The second observation is that the achieved occupancy is also unable to predict the optimal number of SMs. The achieved occupancy of **k-means (1)** drops to 80% around 30 SMs, meaning that the warp scheduler cannot perfectly hide the latency of currently stalled warps. However, for the optimal number of SMs, 25 (as shown in subsection 6.1.1), the achieved occupancy is only 90%. Therefore, achieved occupancy cannot predict **k-means (1)**'s SM-waste.

Thus, occupancy is limited in the amount and the type of waste it can predict, especially at compile-time.

[1] `https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator`

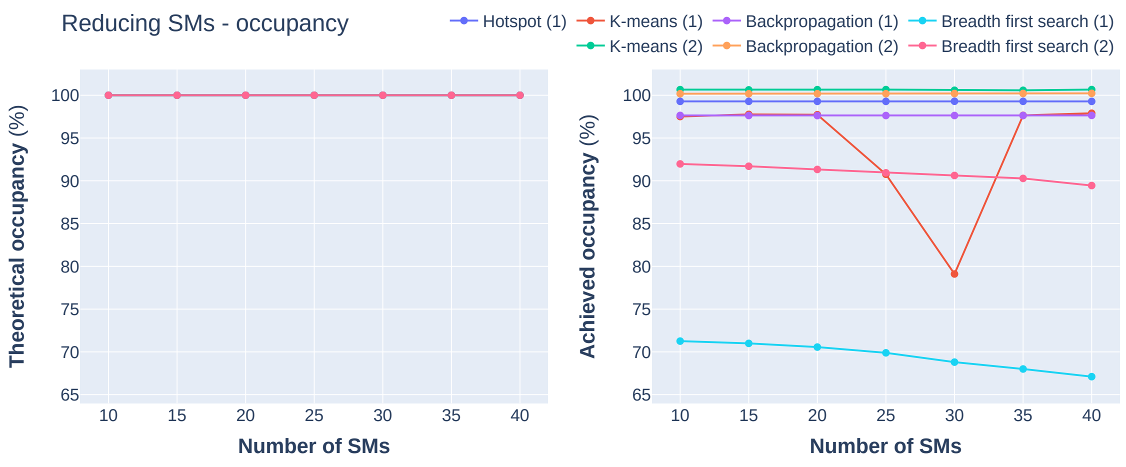

Figure 6.11: Theoretical versus achieved occupancy, for various number of SMs. Theoretical occupancy is calculated using Nvidia Nsight Compute's Occupancy Calculator[1]. Achieved occupancy is calculated as the average active warps relative to the maximum, averaged over all invocations.

7 Conclusion

In this chapter we summarize the main findings of this work, highlight our main contributions, discuss potential limitations, and sketch future work directions.

7.1 Summary and main findings

In our quest to assess the potential (GPU) computing waste, this work proposes a novel definition for computing waste. Intuitively, any surplus of resources that brings no performance gain is an indicator for waste (section 3.1).

Further, we describe a systematic approach to detect such waste at a resource level: we define different system configurations, with different mixes of resources, and compare the performance they achieve when running the same workload. The smallest configuration that achieves a given performance threshold is considered ‘optimal’; all the other configurations lead to waste (chapter 4).

In practical terms, in our work, the exploration starts from configurations of real systems (i.e., GPUs that exist as products) and, using simulation, we create ‘reduced’ GPUs with less SMs, or lower operating frequencies for the memory or the cores. We use these reduced GPUs to run the original workload and measure the difference in execution time and energy consumption. When no performance gain is visible between the original and a reduced GPU, we have detected computing waste (chapter 5).

We evaluate our waste detection workflow on six kernels from the Rodinia [1] benchmark, different GPU configurations that vary SMs, core clock, and memory clock, and, using the Accel-Sim [3] simulator, observe different cases of waste (chapter 6). We further attempt to reconfigure the GPU towards an optimal configuration by combining different types of waste (section 6.3).

7. CONCLUSION

In summary, our main findings are:

- *GPU simulation can be used to predict waste.* In section 6.1 we have demonstrated how simulating reduced configurations of GPU resources can be used to discover waste.

- *Increasing resources will not always yield better performance.* As shown in subsection 6.1.1, reducing the SM-waste for **k-means (1)** will not only lead to a lower energy consumption, but also a lower execution time. Performance engineers and system designers should therefore be extra mindful waste or it can, besides the extra cost, space, and power, also harm performance.

- *Waste scales with workload size.* In section 6.2 we demonstrated how the existence and the amount of waste stays the same across different workload sizes for a both a compute-bound and memory-bound kernel. Although, this experiment is limited to a single resource and a single application, we belief that, due to the nature of GPU computing, this trend is applicable to other resources and applications.

- *Waste cannot be trivially combined.* In section 6.3, we have tried to simultaneously reduce different types of waste, and found that trivially combining (individually) optimal resources will reduce less waste than reducing only one of the resources. In terms of performance, the combined system will still outperform the baseline, but underperform against the best individual system.

- *Arithmetic intensity is correlated with (the type of) waste.* We observed that, in general, waste is more likely to exist in memory-related resources for compute-bound kernels and in compute-related resources for memory-bound kernels. For all kernels, if waste was discovered[1], it was on the opposite side of the dominating instruction type, i.e, we did not observe a situation where *memory-related* resources were wasted in *memory-bound* kernels.

- *AccelWattch cannot accurately predict the power consumption of the core and memory clocks.* As stated in subsection 6.1.2 and subsection 6.1.3, Accel-Sim's power model AccelWattch, does not incorporate the extra static power consumed when increasing

[1]Note that some resources can affect both. For example, SMs provide both compute and memory (L1) power. Similarly, the core clock affects both the latency of compute operations and the latency of L2 requests. We observed that for some memory-bound kernels it can therefore be beneficial not to reduce these, primarily compute-related, resources.

the core or memory clock. Our energy measurements for these resources are therefore overly optimistic for higher clocks. In practice we would expect to find a larger energy benefit when removing the memory clock waste for **hotspot** and **k-means (2)**.

Revisiting our main research goal - **define a workflow for measuring waste in GPU-based applications** - we have partially achieved it: our simulation-based analysis does enable the identification of GPU computing waste (in alignment with our definition), but we did not succeed in providing a quantitative measure of waste.

7.2 Contributions

The main contribution of this thesis are:

- We propose a first operational definition for computing waste:

  Definition 2 (system waste)**.** *Equal or better performance with a reduced system and a fixed application and workload.*

  We use this definition because (i) if it is true (for some resource), it will always be considered unfavourable, (ii) it is independent of the to-be-reduced resource (as only the performance counts), and because (iii) any interpretation of, or decision on, the underlying utilization or efficiency metrics can never inadvertently harm performance because we ultimately measure waste in terms of performance itself. Using this definition, we have successfully found SM, core clock, and memory clock waste. In subsection 6.1.4, we experimented with a relaxed definition that allows for a certain performance decrease in return for fewer resources. We find that a relaxed definition can identify both new waste and larger amounts of the same waste. We also observe that, generally, the amount of extra waste found decreases with the size of the tolerated performance penalty.

- We define a workflow for assessing computing waste in GPU-based applications. We outlined the workflow in chapter 4 and, using the experiments from chapter 5, demonstrated it in chapter 6. Although acquiring the metrics needed is time-intensive, the setup and the analysis is fast and intuitive. We believe our workflow can be used for most (GPU) resources and applications. Furthermore, our workflow is agnostic to the underlying performance model. To identify waste, the only metrics needed are the number of cycles and the (average) power consumption. Our workflow is therefore not inherently bounded by the limitations and inaccuracies of any performance model.

- We confirm a pattern correlating waste and arithmetic intensity. Our analysis has shown that memory-intensive kernels exclusively waste compute-related resources, and vice versa.
- We identify several instances of waste in the Rodinia [1] benchmark. Specifically, we found SM and core clock waste for the first K-means kernel, and memory clock waste for the second K-means kernel and the hotspot kernel.
- We empirically show that more diversity in GPU configurations can reduce compute waste, but finding a one-size-fits-all is non-trivial.

7.3 Limitations and threats to validity

We have identified the following limitations and threats to validity for our workflow, experimental setup, and evaluations:

- *The accuracy of our workflow can only be as good as the accuracy of the underlying performance model.* Specifically, the amount and presence of waste we discovered in chapter 6 is bounded by the accuracy of the underlying simulator, Accel-Sim. Like all models, the accuracy of simulator will decline as the configurations stray further from realistic, potentially validated, system configurations. Thus, the more we reduce resources (or combine multiple reduced resources) the less trustworthy the simulator, and therefore our workflow, will be. Further note that any inaccuracies between the performance model and the actual hardware will not always be constant. As shown in subsection 6.1.2, the simulator incorrectly predicts the trend in power when varying the core and memory clock.

  Moreover, the performance model can be limited in scope or speed. For instance, Accel-Sim cannot simulate streams or a heterogeneous (CPU-GPU or GPU-GPU) workload.

- *Reliance on averages.* In our analysis we implicitly or explicitly rely on averages or summations of the execution results. As an example, we measure the *average* power consumption, but the system must be able to provide the *maximum* power (which, for the optimal hotspot system, can be an up to 50 Watt difference). Likewise, the efficiency metrics are first averaged over the execution of each invocation, and then averaged over all invocations, even though the efficiency (and utilization) will fluctuate between different moments in the execution. For example, most kernels follow an "compute indexes $\rightarrow$ load data $\rightarrow$ compute result $\rightarrow$ store result" pattern. Each section will likely favour completely different resources (thereby creating waste in others). However, because our goal is to discover waste on a kernel-level, we focus only on the, on average, dominating operations. Furthermore, because we measure waste using (the total) performance, unrepresentative averages can only hinder our understanding.

- *Ability to implement.* For the sake of demonstration, we do not limit ourselves to available hardware or common hardware limitations in our experiments. However, in practice, one should incorporate the limitations of contemporary or future hardware. For instance, as shown in Figure 6.4, the selection of configurable memory clocks in hardware is smaller and more specific than our simulated range. Second, the hardware continuously matches the clocks to the demand, meaning that a fixed clock is unrealistic. Lastly, all hardware is power capped, meaning that the system must always make do with a certain maximum amount of power.

  Even though a full list of hardware limitations and/or preferences does not exist (and will inevitability change), it could help prune the number of considered configurations and make the results more realistic.

7.4 Future work

We consider the following promising directions for extending our workflow or analysis.

- *Quantify waste.* Our definition of waste does not specify a metric in which waste can measured. We currently quantify waste as the number of reduced resources along with the amount of performance gained to denote its significance. Unfortunately, we have not found a metric that can combine the two. Specifically, when measuring waste we currently face the following dilemmas:

  - *Comparing and combining waste per (performance) metric.* As shown in subsection 6.1.4, it could happen that a kernel yields conflicting results for different metrics, i.e., when reducing resources, the time increases while the energy decreases or vice versa. Furthermore, if waste exists for more than one metric (but at different resource amounts), we cannot compare their difference nor decide the 'superior waste'. Therefore, we have limited our analysis to waste for a single metric at a time.

  - *Comparing and combining waste per resource.* In section 6.3 we have experimented with combining waste for different resources simultaneously (e.g. core clock and SM waste). We have found that waste is not additive (as the combined system performed worse), and can therefore not be trivially combined. Although bounded by the individual waste per resource, the optimal system will reduce an unknown combination of these different types of waste.

  A metric-agnostic and resource-agnostic waste metric or classification could help to compare, combine, and show the impact of waste. Alternatively, to combine waste for different resources, future work could devise a better strategy (e.g., resource-by-resource) or a more efficient way to traverse the design space (i.e., all possible (combined) configurations).

- *Define waste trends* As stated in section 7.1, we discovered that the arithmetic intensity, i.e., the ratio between compute and memory operations, will, on a high-level, predict which resources to reduce. However, arithmetic intensity is unlikely to predict finer-grained resources. This is because, on the memory side, it cannot incorporate the ratio between (or the hit rate of) the L1, L2, and the memory traffic. Similarly, on the compute side, it cannot differentiate between different operations of the same type (e.g. double-precision) nor combine operations of different types. Therefore, to predict finer-grained resources, more detailed indicators are needed, e.g., x L2 lines at y bytes/s are used for every z integer instructions.

  A related direction is characterizing applications and their respective waste in order to generalize waste reduction strategies.

- *A cheaper performance model.* Our workflow uses a GPU simulator to calculate the performance metrics needed to find waste. However, as shown in section 3.2, simulation is relatively heavyweight solution, i.e., for hotspot, the simulation took 5M $\times$ longer to simulate than to run on the hardware directly. This slowdown limits the number of resources and the number of steps per resource that can be tested, thereby limiting the scope and accuracy of waste that can be found. Because of the accuracy, we still believe simulation is, currently, the best option for finding waste. However, to make finding waste practical, there is a need for faster simulators or more accurate analytical or statistical performance models.

- *A direct waste model.* Currently, our workflow for measuring waste, as defined in Figure 4.1, requires a performance model to make predictions. A direct waste model however could predict the same waste directly from the input, i.e. the application, workload, and configurations. Because of the complexities in predicting performance, it is unclear if this is possible, let alone accurate. Furthermore, an analytical or statistical waste model would require more detailed knowledge on the relationships between the application/workload and the resulting waste or a learning set large enough to infer these relationships accurately, respectively.

- *Increasing resources.* Our definition of waste only considers reducing resources (from some arbitrary baseline). However, we suspect there could be a situation where adding more resources of one kind will reduce (the total) waste for others. For example, increasing the core clock while reducing the number SMs so that the same amount of compute is possible in the same time span might increase performance, possibly due to an increase in energy efficiency or because of the extra L2 throughput. More research is needed to analyze the possibilities of this technique and adjust or expand the definition accordingly.

References

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. **Rodinia: A benchmark suite for heterogeneous computing**. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. ii, 2, 10, 21, 32, 59, 62

[2] John Shalf, Dan Quinlan, and Curtis Janssen. **Rethinking Hardware-Software Codesign for Exascale Systems**. *Computer*, **44**(11):22–30, 2011. 2, 19, 20, 21, 23

[3] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. **Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling**. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020. 5, 19, 26, 27, 31, 43, 59

[4] Richard Schoonhoven, Ben van Werkhoven, and K Joost Batenburg. **Benchmarking optimization algorithms for auto-tuning GPU kernels**. *IEEE Transactions on Evolutionary Computation*, 2022. 9

[5] Ben van Werkhoven. **Kernel Tuner: A search-optimizing GPU code auto-tuner**. *Future Generation Computer Systems*, **90**:347–358, 2019. 9

[6] Nvdia. **NVIDIA Multi-Instance GPU**. 20

[7] Samuel Williams, Andrew Waterman, and David Patterson. **Roofline: an insightful visual performance model for multicore architectures**. *Commun. ACM*, **52**(4):65–76, apr 2009. 1, 20, 26

[8] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016. 20

[9] Jounghoo Lee, Yeonan Ha, Suhyun Lee, Jinyoung Woo, Jinho Lee, Hanhwi Jang, and Youngsok Kim. **GCoM: a detailed GPU core model for accurate analytical modeling of modern GPUs**. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 424–436, New York, NY, USA, 2022. Association for Computing Machinery. 21

[10] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and José Carlos Sancho. **Using Performance Modeling to Design Large-Scale Systems**. *Computer*, **42**(11):42–49, 2009. 21

[11] Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, and John Shalf. **ExaSAT: An exascale co-design tool for performance modeling**. **29**(2):209–232, may 2015. 21, 25

[12] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. **Stargazer: Automated regression-based GPU design space exploration**. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 2–13, 2012. 11, 21

[13] Kenneth O'Neal and Philip Brisk. **Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey**. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 763–768, 2018. 21

[14] Chunrong Yao, Wantao Liu, Weiqing Tang, and Songlin Hu. **EAIS: Energy-aware adaptive scheduling for CNN inference on high-performance GPUs**. *Future Gener. Comput. Syst.*, **130**(C):253–268, may 2022. 22

[15] G. De Michell and R.K. Gupta. **Hardware/software co-design**. *Proceedings of the IEEE*, **85**(3):349–365, 1997. 22

[16] Mitsuhisa Sato, Yuetsu Kodama, Miwako Tsuji, and Tesuya Odajima. **Co-Design and System for the Supercomputer "Fugaku"**. *IEEE Micro*, **42**(2):26–34, 2022. 22

[17] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. **Statistical GPU power analysis using tree-based methods**. In *2011 International Green Computing Conference and Workshops*, pages 1–6, 2011. 10

[18] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. **Analyzing CUDA workloads using a detailed GPU simulator**. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009. 10, 12, 27

[19] Yuki Abe, Hiroshi Sasaki, Martin Peres, Koji Inoue, Kazuaki Murakami, and Shinpei Kato. **Power and performance analysis of GPU-accelerated systems**. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, page 10, USA, 2012. USENIX Association. 10

[20] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. **A characterization and analysis of PTX kernels**. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 3–12, 2009. 10

[21] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Liu, and Wen mei W. Hwu. **Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing**. 2012. 10

[22] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. **A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads**. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11, 2010. 11, 12

[23] Molly A. O'Neil and Martin Burtscher. **Microarchitectural performance characterization of irregular GPU kernels**. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 130–139, 2014. 11, 13

[24] Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li. **Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications**. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–10, 2010. 11

[25] Pablo Carvalho, Lúcia M. A. Drummond, Cristiana Bentes, Esteban Clua, Edson Cataldo, and Leandro A. J. Marzulo. **Analysis and Characterization of GPU Benchmarks for Kernel Concurrency Efficiency**. In Esteban Mocskos and Sergio Nesmachnow, editors, *High Performance Computing*, pages 71–86, Cham, 2018. Springer International Publishing. 11

[26] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. **The Scalable Heterogeneous Computing (SHOC) benchmark suite**. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery. 11

[27] Behzad Boroujerdian, Ying Jing, Devashree Tripathy, Amit Kumar, Lavanya Subramanian, Luke Yen, Vincent Lee, Vivek Venkatesan, Amit Jindal, Robert Shearer, and Vijay Janapa Reddi. **FARSI: An Early-stage Design Space Exploration Framework to Tame the Domain-specific System-on-chip Complexity**. *ACM Trans. Embed. Comput. Syst.*, **22**(2), jan 2023. 12

[28] Changxi Liu, Yifan Sun, and Trevor E. Carlson. **Photon: A Fine-grained Sampled Simulation Methodology for GPU Workloads**. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1227–1241, New York, NY, USA, 2023. Association for Computing Machinery. 12

[29] Lu Wang, Magnus Jahre, Almutaz Adileho, and Lieven Eeckhout. **MDM: The GPU Memory Divergence Model**. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1009–1021, 2020. 12

[30] Robin L Plackett and J Peter Burman. **The design of optimum multifactorial experiments**. *Biometrika*, **33**(4):305–325, 1946. 13

[31] Gingfung Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. **Towards GPU utilization prediction for cloud deep learning**. In *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'20, USA, 2020. USENIX Association. 13

[32] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. **Analyzing CUDA workloads using a detailed GPU simulator**. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009. 13

[33] Sina Darabi, Mohammad Sadrosadati, Negar Akbarzadeh, Joël Lindegger, Mohammad Hosseini, Jisung Park, Juan Gómez-Luna, Onur Mutlu,

and Hamid Sarbazi-Azad. **Morpheus: Extending the Last Level Cache Capacity in GPU Systems Using Idle GPU Core Resources**. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–244, 2022. 13

[34] Yaser Jararweh, Shadi Alzubi, and Salim Hariri. **An optimal multi-processor allocation algorithm for high performance GPU accelerators**. In *2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–6, 2011. 14

[35] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. **MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters**. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 173–189, New York, NY, USA, 2022. Association for Computing Machinery. 14

[36] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. **Dynamic Resource Management for Efficient Utilization of Multitasking GPUs**. *SIGPLAN Not.*, **52**(4):527–540, apr 2017. 14

[37] Tyler Yandrofski, Jingyuan Chen, Nathan Otterness, James H. Anderson, and F. Donelson Smith. **Making Powerful Enemies on NVIDIA GPUs**. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 383–395, 2022. 14

[38] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. **Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator**. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 868–880, 2021. 26

[39] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. **The gem5 simulator**. *ACM SIGARCH computer architecture news*, **39**(2):1–7, 2011. 26

[40] Bradford M Beckmann and Anthony Gutierrez. **The AMD gem5 APU simulator: Modeling heterogeneous systems in gem5**. In *Tutorial at the International Symposium on Microarchitecture (MICRO)*, 2015. 26

[41] HSA Foundation. **HSA Programmer's Reference Manual 1.2**. 26

[42] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. **MGPUSim: enabling multi-GPU performance modeling and optimization**. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 197–209, New York, NY, USA, 2019. Association for Computing Machinery. 27

[43] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. **Macsim: A cpu-gpu heterogeneous simulation framework user guide**. *Georgia Institute of Technology*, 2012. 27

[44] Prasun Gera, Hyojong Kim, Hyesoon Kim, Sunpyo Hong, Vinod George, and Chi-Keung Luk. **Performance Characterisation and Simulation of Intel's Integrated GPU Architecture**. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 139–148, 2018. 27

[45] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. **Multi2Sim: A simulation framework for CPU-GPU computing**. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344, 2012. 27

[46] Xun Gong, Rafael Ubal, and David Kaeli. **Multi2Sim Kepler: A detailed architectural GPU simulator**. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 269–278. IEEE, 2017. 27

[47] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. **AccelWattch: A Power Modeling Framework for Modern GPUs**. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 738–753, New York, NY, USA, 2021. Association for Computing Machinery. 27, 43

[48] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. **McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures**. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, pages 469–480, 2009. 27

[49] Michael Boyer. *Improving Resource Utilization in Heterogeneous CPU-GPU Systems.* PhD thesis, Computer Engineering, School of Engineering and Applied Science, University of Virginia, April 2013. 43