Improving GPGPU Energy-Efficiency through Concurrent Kernel Execution and DVFS

Qing Jiao[1] Mian Lu[2] Huynh Phung Huynh[2] Tulika Mitra[1]

[1]National University of Singapore [2]Institute of High Performance Computing, A*STAR

jiaoqing@comp.nus.edu.sg {lum,huynhph}@ihpc.a-star.edu.sg tulika@comp.nus.edu.sg

Abstract

Current generation GPUs can accelerate high-performance, compute-intensive applications by exploiting massive thread-level parallelism. The high performance, however, comes at the cost of increased power consumption. Recently, commercial GPGPU architectures have introduced support for concurrent kernel execution to better utilize the computational/memory resources and thereby improve overall throughput. In this paper, we argue and experimentally validate the benefits of concurrent kernels towards energy-efficient execution. We design power-performance models to carefully select the appropriate kernel combinations to be executed concurrently, the relative contributions of the kernels to the thread mix, along with the frequency choices for the cores and the memory to achieve high performance per watt metric. Our experimental evaluation shows that the concurrent kernel execution in combination with DVFS can improve energy-efficiency by up to 34.5% compared to the most energy-efficient sequential execution.

1. Introduction

Current generation GPUs are well positioned to satisfy the computational requirements of high-performance applications. Starting from fixed-function graphics pipeline to a programmable massive multi-core for advanced realistic 3D graphics [6] and accelerators for general purpose applications, the performance of GPUs has progressed at a phenomenal rate in the past two decades exceeding the projection of Moore's Law [13]. For example, NVIDIA GeForce GTX TITAN Z GPU has a peak performance of 8 TeraFLOPS [2] and AMD Radeon R9 has a peak performance of 11.5 TeraFLOPS [1]. The high performance of the GPUs comes at the cost of high density of computational resource on a single chip. With the failure of Dennard Scaling [7], the power density and total power consumption of GPUs have escalated rapidly. Hence, power management of GPUs have become increasing important.

There are many different approaches to power management of GPUs, starting from circuit/architecture level all the way to the software level. In the commercial space, AMD and NVIDIA have included power management of the GPUs in recent years. AMD uses PowerPlay to reduce the dynamic power consumption. Based on the utilization of the GPU, PowerPlay puts the GPU into low, medium and high power states accordingly. Similarly, NVIDIA uses PowerMizer to reduce power. Dynamic Voltage and Frequency Scaling (DVFS) is the most widely used mechanism for power management due to its ease of implementation and significant payoff in terms of energy-efficiency. DVFS can be exploited in software layers in various ways. For example, Lee et al. [11] and Jiao et al. [10] attempt to change the frequency of the GPU core and the memory based on the compute- and memory-intensity of the computational kernel running on the GPU.

The new generation of GPUs, such as NVIDIA Fermi and Kepler series GPUs, support concurrent kernel execution. A single kernel may not be able to utilize all the resources available in a GPU. Concurrency enables the execution of multiple kernels simultaneously in the GPU. If the cohabitant kernels are complementary in nature in terms of their compute- and memory-intensity, then concurrent execution of the kernels leads to better utilization of the resources and improved throughput. For example, Zhong et al. [16] exploit the kernel features to select kernels with complementary memory and compute intensity to run concurrently, so as to improve the GPU throughput. Moreover, the under-utilization of the resources during execution of a single kernel leads to unnecessary wastage of power. The better utilization of the resources with concurrent kernels combined with improved throughput can potentially lead to significantly better energy-efficiency (see the motivating example in Section 2). However, to the best of our knowledge, currently there does not exist any work that attempt to exploit the concurrency to improve the GPU energy efficiency. In this paper, we explore a combination of concurrent execution of kernels and DVFS to improve the performance-per-watt behavior compared to the default sequential model of execution.

Even through current generation GPUs support concurrency, their scheduling policies allow only minimal overlap in execution among the kernels [14] and degenerates to almost sequential execution of the kernels in most cases. We overcome this restriction by applying a software-level approach called kernel slicing [16] to enable true concurrency among the kernels. Each participating kernel is partitioned into slices (consisting of a number of blocks) and the slices from the different kernels are interleaved in the CUDA program to force the GPU to execute them in a concurrent manner.

In order to extract the maximum benefit from the concurrent execution, we need to (a) select the subset of kernels that should execute together, (b) determine the proportion of each kernel in the mix, and (c) tune the core and memory clock frequency settings appropriately so as to obtain the maximum performance-per-watt benefit. We develop accurate power-performance estimation models to help us make these design choices at runtime quite efficiently. A greedy scheduling algorithm takes advantage of these estimation models to choose appropriate kernel combinations to execute from a waiting pool of kernels. Our experimental evaluation on a contemporary GPU platform reveals up to 34.5% improvement in performance per watt due to concurrent execution.

This rest of the paper is organized as follows. We present a motivating example to impress the importance of concurrent execution towards energy-efficiency in Section 2. We describe our experimental setup in Section 3. The concurrent execution of kernels through kernel slicing is presented in Section 4 followed by the scheduling algorithm in Section 5. The heart of our approach is the power-performance estimation model, which appears in Section 6.

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00 ©2015 IEEE

We evaluate our approach in Section 7 and present related work in Section 8. Finally we conclude in Section 9.

2. Motivating Example

In this work, our goal is to optimize the performance per watt metric through concurrent kernel execution and DVFS. As our target applications contain a mix of integer and floating point operations, we use GOPS (Giga Operations Per Second) instead of FLOPS (FLoating Point Operations Per Second) as the performance metric. We use **GOPS/Watt** as the metric for energy efficiency that captures the computational capability for every Watt of power consumed by the GPU.

Let us now proceed to illustrate the energy-efficiency benefits of concurrent kernels and DVFS through a simple motivating example. For this example, we choose two benchmark programs *Hotspot* and *Mergehist*. The details of the experimental setup will be presented in Section 3.

Execution scenario	GOPS/Watt
Concurrent optimal block ratio 3:5, optimal freq	247
Concurrent optimal block ratio 3:5, max freq	217
Concurrent block ratio 4:4, optimal freq	219
Concurrent block ratio 4:4, max freq	206
Sequential optimal freq	177
Sequential max freq	166

Table 1: Impact of concurrency, frequency setting, and block ratio on energy efficiency

Traditionally, the kernels will be executed sequentially. We make two different choices regarding the frequency level of the GPU and the memory during the sequential execution of the kernels: (a) run both the kernels at the maximum frequency level of the GPU and the memory, (b) tune the GPU, memory frequencies for each individual kernel to achieve optimal GOPS/Watt during the execution of that kernel. In the second scenario, the two kernels might run at different frequency levels as each kernel runs at its optimal frequency.

The concurrent execution of the kernels we employ in this work involves executing blocks from all the kernels in parallel in a fixed ratio. We call the proportion of each kernel in this kernel mix as the **block ratio**. For example, a block ratio of 3:5 for kernels *Hotspot* and *Mergehist* implies that 3 blocks of *Hotspot* execute in parallel with 5 blocks of *Mergehist* at any point in time. We identify the block ratio of the two kernels to achieve the optimal energy-efficiency through exhaustive search. The optimal block ratio for our example kernels is *Hotspot:3, Mergehist:5*. Similar to the sequential execution, we first run the concurrent kernel with the optimal block ratio at maximum frequency level. Next, we identify the most energy-efficient frequency for the concurrent kernel with optimal block ratio and run it at that frequency.

Table 1 shows the GOPS/Watt of the sequential and concurrent execution of the two kernels. The concurrent execution improves the energy efficiency by 39% over the sequential execution. Furthermore, the figure confirms that running the concurrent kernel at the maximum frequency setting does not achieve the best energy efficiency. The concurrent execution at the highest frequency setting has a lower 217 GOPS/Watt compared to 247 GOPS/Watt at the optimal frequency setting. To conclude, we make two important observations from this example: *(a) concurrent execution can improve energy efficiency significantly over sequential execution, and (b) tuning the GPU, memory frequency is crucial to achieve the best energy efficiency.*

Clearly, improving energy efficiency involves finding the optimal block ratio and the optimal frequency setting. A natural question that arises is the choice of the block ratio for different objectives. For example, whether the block ratio with performance as the objective is the same as the block ratio for energy efficiency as the objective. To answer this question, we first identify the optimal block ratio for highest performance and it turns out to be *Hotspot:4, Mergehist:4*. Running the concurrent kernel at 4:4 ratio and highest frequency results in 206 GOPS/Watt (see Table 1). We further optimize this execution to 219 GOPS/Watt by running it at the optimal frequency for energy efficiency. Still the energy efficiency at 4:4 block ratio is much lower than 247 GOPS/Watt at 3:5 block ratio. Clearly, the selection of the block ratio needs to take performance per watt objective into account rather than performance alone. This observation differentiates our work from the performance orientated GPU work in terms of choosing kernel combination. Moreover, this experiment also establishes that *the appropriate block ratio selection is imperative to optimal energy efficiency.*

3. Experiment Setup

Before we proceed further, it is important to introduce our experimental setup in this section.

3.1 Experimental Platform

Core Clock (MHz)	Memory Clock (MHz)
562	324
705	400
836	480
967	550
1097	625
1228	710

Table 2: Core and memory clock frequency levels

We conduct all our experiments and analysis on NVIDIA GeForce GT 640 graphics card. GeForce GT640 features Kepler GPU architecture, the most widely used GPU architecture at this point. This particular version of the graphics card consists of two SMX (Streaming Multiprocessors) and 2GB DRAM memory. Each SMX contains 192 CUDA cores. Both the core and the memory clock can be set to six discrete frequency levels. Table 2 shows the frequency settings. Therefore, there are 36 pairs of core and memory frequency combinations. Increasing the clock speed of the core improves the processing capability, while increasing the memory clock speed improves the memory bandwidth. On the other hand, increasing both the core and the memory clock speed has negative impact on power. We change the frequency setting using third-party software EVGA PrecisionX 16. The voltage is kept constant at 1.012V for all the experiments.

We use NVIDIA Visual Profiler for performance measurement and to obtain detailed performance counter information from the GPU. We use National Instrument SC-2345 DAQ (Data Acquisition) system along with PCI EXPRESS Bus Extender for power measurement of the entire GPU platform. We plug in the PCI extender in the original PCI slot for the GPU and then plug in the GPU in the PCI extender. This ensures that the power supply for the GPU goes through the PCI extender. The current difference between two specific points of the PCI extender is be the current consumed by the GPU. As we know the supply voltage and the resistor parameter, we can compute the GPU power consumption. National Instrument SC-2345 is used to measure the current difference between the specified points.

Note that the high-end GPUs will contain more SMX compared to GeForce GT 640. If a GPU contains more SMX, we expect to save more energy for memory-intensive kernels with concurrency (as more computation become available through concurrency). In general, for any GPU platform, different kernel combinations have

different energy efficiency improvements. Therefore, we can always expect kernel combinations enabled by concurrent execution will have significant energy efficiency improvement over sequential executions irrespective of the GPU platform.

3.2 Benchmarks

Kernel	GOPS	Memory bandwidth (Gbytes/sec)	No of blocks
Pathfinder	7.9	1.9	1300
Bitonic	4.6	19.3	5000
Bt	10.1	0.1	500
Hotspot	7.7	0.5	10000
Layer	9.2	1.8	3600
Samplerank	4.0	17.5	3000
Srad	5.3	19.5	5000
Matrix	9.9	0.6	500
Time_step	2.8	18.8	16000
Mergehist	4.2	0.8	5000
Transpose	7.7	13.9	16000

Table 3: Benchmark kernels' characteristics at highest frequency

We choose 11 real-world kernels with various compute- and memory intensity as benchmarks for our experiments. The kernels *Bitonic, Samplerank, Matrix* and *Mergehistogram* are selected from CUDA Sample 5.5. The rest of the benchmarks are selected from Rodinia Benchmark suite 2.4 [4]. The characteristics of these kernels while running at the highest core and clock frequency levels are shown in Table 3. The table presents the compute requirements of the kernels as GOPS (Giga Operations Per Second), while the memory bandwidth requirement in Gbytes/sec (Gigabytes per second). The input size of the kernel is determined by the number of blocks present in the kernel. As we can see from the table, some kernels are compute intensive (e.g., *Bt*), some are memory intensive (e.g., *Time_step*), and some others require both compute and memory in equal measures (e.g., *Transpose*).

4. Concurrent Kernel Execution

In this section, we detail the mechanisms for concurrent kernel execution. We employ NVIDIA Kepler architecture as the representative GPU platform for execution of concurrent kernels. The GPU in Kepler architecture consists of several powerful SMX (Streaming Multiprocessor) that share an L2 cache and the main memory. Each SMX contains 192 single-precision CUDA cores, 64 double-precision units, 32 special function units, and 32 load/store units. The CUDA cores and the other function units within an SMX share 64 KB on-chip memory that can be configured as shared memory or L1 cache and 65,536 entry register file.

In CUDA, a computational kernel comprises of hundreds or thousands of threads operating on different data in parallel. The threads are organized into warps and blocks. Every 32 threads are organized into one warp. Warps are further grouped into blocks. GPU block scheduler dispatches blocks to the different SMX. The number of blocks that could be dispatched into an SMX depends on the resource usage of the blocks, such as the number of warps, shared memory size and register usage. Warp is the scheduling unit within SMX. Each SMX features four wrap schedulers that can issue four wraps simultaneously to the CUDA cores each cycle. All the threads within a warp execute simultaneously processing different data elements. The block and warp scheduling policies determine the order of execution of the threads in concurrent kernels.

Block scheduler: The block scheduler allocates the blocks to different SMX in a balanced way. When a block is ready to be scheduled, the block scheduler first calculates the available resources on each SMX, such as free shared memory, registers, and number of warps. The block is then scheduled to the SMX with the maximum available resources. The current GPU scheduler employs leftover policy for concurrent kernels [14]. Leftover policy is essentially equivalent to sequential execution. In this policy, the scheduler first dispatches all the blocks from one of the kernels, say K_1, in the mix. If there are available resources in an SMX after all the blocks from the kernel K_1 have been dispatched, then the blocks from the following kernel K_2 are dispatched and this results in concurrent execution of the two kernels. That is, concurrent execution may only happen in the overlapping region at the end of a kernel and the beginning of the next kernel. Therefore, we use kernel slicing [16] to accomplish true concurrency among multiple kernels in commercial GPU architecture.

Warp scheduler: The Kepler architecture supports different kernels running concurrently within one SMX. After the block scheduler schedules the blocks to the SMX, an SMX may contain blocks from different kernels. Unfortunately, the inner working of the warp scheduler in the Kepler architecture is not available from the literature. Therefore, we use micro-benchmarks to verify that the four warp scheduler within an SMX can indeed schedule warps from different kernels in the same cycle.

We first create a simple CUDA kernel consisting of only integer operations. The kernel consists of 16 blocks and each block comprises of only one warp. During the execution of the kernel, the four warp schedulers within the SMX dispatch four warps per cycle to the CUDA cores to fully utilize the computational resources.

Next we create 16 CUDA kernels, each consisting of only one block identical to the blocks in the single-kernel version and the block comprises of only one warp. In other words, the total computation for both single-kernel and multi-kernel versions are identical. Clearly, under the leftover policy, different kernels will be dispatched to each SMX to fully exploit the available resources. If the four warp schedulers cannot schedule warps from different kernels per cycle, then only one warp can execute per SMX at any point in time and the performance of the multi-kernel version will be much worse than the performance of the single-kernel version. However, we observe that the runtime of multi-kernel version is almost the same as that of the single-kernel version. This experiments substantiates the claim that the warp schedulers can indeed schedule warps from different kernels in the same cycle.

Kernel slicing: As mentioned earlier, although GPUs support concurrent kernel execution, the leftover policy of block scheduling only allows minimal overlap among the blocks from different kernels. Therefore, we choose to use kernel slicing [16] and CUDA streams to accomplish concurrency under the leftover block scheduling policy. Kernel slicing divides the thread blocks of a kernel into multiple slices. The concept of kernel slicing is easy to illustrate with an example code.

The code fragment in Algorithm 1 shows the default CUDA code for execution of two kernels K_1 and K_2. Each kernel consists of 100 blocks where each block has only one warp. Figure 1 shows the scheduling of the blocks from the two kernels under leftover policy. Each SMX can support 16 blocks of K_1 at any point in time. In the end, we are left with 4 blocks of K_1. Hence the leftover policy starts scheduling blocks of K_2 to utilize the leftover resources. As we can see from Figure 1, the overlap of execution between the two kernels is quite minimal in the default policy.

Algorithm 1 Default CUDA code for two kernels

K_1 <<<100, block size,streams[0] >>> (function parameters);
K_2 <<<100, block size,streams[1] >>> (function parameters);

Algorithm 2 CUDA code for kernel slicing

```
K1 <<<6,  block size,streams[0]>>> (function parameters);
K2 <<<10, block size,streams[1]>>> (function parameters);
K2 <<<10, block size,streams[2]>>> (function parameters);
K1 <<<6,  block size,streams[3]>>> (function parameters);
K2 <<<10, block size,streams[4]>>> (function parameters);
...
```

Now suppose we have identified $K_1 : 6 \;\; K_2 : 10$ as the optimal block ratio for the execution of the two kernels. In kernel slicing, the number of blocks per kernel slice is determine by this block ratio. Therefore, we have to create 6-blocks kernel slices for K_1 and 10-blocks kernel slices for K_2. The information about kernel slices are communicated to the GPU through CUDA streams. Algorithm 2 shows the CUDA code with kernel slicing. Our goal is to execute 6 blocks from K_1 concurrently with 10 blocks from K_2. This is easy to achieve in the beginning. However, the blocks from K_2 finish execution earlier. Then we need to feed the SMX with blocks from K_2 to keep the block ratio at 6:10. Thus in the CUDA code, there are two consecutive calls to process kernel slices of K_2 before a call to process kernel slice of K_1. Figure 2 shows the scheduling of the blocks from the two kernels with kernel slicing. Note that we may need to change the block index in the kernel function to support kernel slicing as shown in previous studies [14]. Partitioning a kernel into multiple slices will cause the CPU to issue more system calls to the GPU. However, compared to the block running time (mostly at millisecond or microsecond level) this system call overhead can be ignored as we observed in our experiments. Still our experimental results include the overhead of these additional system calls for concurrent execution.

It is straightforward to generate the CUDA code with kernel slicing from the default code. In our experiment, we first measure the execution time of each kernel block. Then Algorithm 3 is used to automatically generate the correct slice order. We estimate the kernel slice that will complete execution first based on its block execution time and the number of blocks in the block ratio. We simply dispatch another slice of that kernel to the GPU. Once all the blocks of a constituent kernel have been processed, the concurrent kernel no longer has all its members and hence we do not allow the concurrent kernel to proceed further. The incomplete kernels in the concurrent kernel mix will proceed execution presumably with a different mix of kernels.

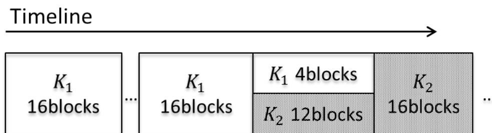

Figure 1: Kernel execution under leftover policy.

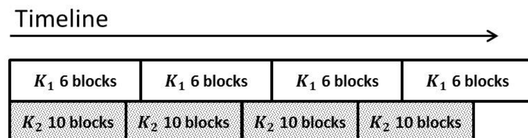

Figure 2: Concurrent kernel execution with kernel slicing.

5. Concurrent Kernel Scheduling

The goal of our work is to schedule multiple kernels on the GPU concurrently so as to improve the energy-efficiency of the execution. We assume that there exist multiple kernels waiting to be processed at any point in time. Traditionally, these kernels will be processed in some sequential order on the GPU. But we have already established that concurrent execution of kernels with complementary features (such as memory-intensive and compute-intensive kernels) can significantly improve the energy-efficiency compared to the sequential execution. Therefore, we select an appropriate subset of the kernels to be scheduled together and then tune the core, memory clock frequency for the fused kernel so as to achieve the most energy-efficient execution. As we will show later, combining more than two kernels does not provide enough benefit to warrant the complexity — both in selection and execution. Therefore, we restrict ourselves to two-kernel combinations in most of this work. However, our approach can be easily extended to concurrent execution of more than two kernels with minimal modification.

Algorithm 3 Kernel Slice Order

> T_i := block execution time of K_i; // **input.**
> n_i := No of blocks per kernel slice of K_i; // **input.**
> **for** each kernel **do**
> $ET_i := n_i \times T_i$; //end time of kernel slice from K_i
> **end for**
> **while** $true$ **do**
>   Find the kernel K_j with minimum ET_j
>   **if** there are more kernel blocks from kernel K_j **then**
>   Run kernel slice from kernel K_j ;
> $ET_j = ET_j + n_j \times T_j$;
>   **else**
>   break; //this concurrent kernel is done
>   **end if**
> **end while**

Our approach to kernel scheduling has three components: (a) an offline model for energy-efficiency estimation of concurrent execution of a pair of kernels with a specified block ratio, and (b) an offline model for energy-efficiency estimation of sequential execution of a pair of kernels, and (c) an online scheduling algorithm that picks an appropriate pair of kernels to schedule on the GPU from the waiting pool of kernels. The offline estimation models will be presented in Section 6. In this section, we assume that the estimation models can provide us with the necessary information and we focus only on the online scheduling algorithm.

The online scheduling algorithm is invoked when either (a) a new kernel is added to the waiting pool, or (b) a kernel completes execution. The algorithm performs the following steps to identify the kernel pair with the maximum improvement in energy-efficiency through concurrent execution compared to sequential execution.

1) For every pair of kernels in the waiting pool, we consider all possible block ratios. The Kepler architecture can support at most 16 blocks per SMX. Therefore, we only need to consider all possible 2-part integer partitions of 16 (1+15, 2+14, ...) to cover the different block ratios. As there are 16 such partitions, it is feasible to exhaustively estimate the energy-efficiency of all block ratios.
2) For every pair of kernels and block ratio, we estimate its optimal GOPS/Watt and the corresponding frequency. This step also gives us the optimal block ratio and frequency for the kernel pair and the corresponding GOPS/Watt.
3) For every pair of kernels, we estimate its optimal GOPS/Watt if the kernels execute sequentially.
4) Comparing optimal GOPS/Watt between concurrent and sequential execution, we estimate the performance per watt improvement for each kernel pair through concurrency.

5) We sort the kernel pairs in descending order of GOPS/Watt improvement through concurrency over sequential execution.

The end result in a table similar to Table 4 that captures the optimal GOPS/Watt improvement for concurrent execution of each kernel pair (compared to sequential execution) and the corresponding optimal block ratio and the optimal core, memory frequency settings. This table is updated when a new kernel joins the pool or an existing kernel completes execution. Assuming we have n kernels in the beginning, we have to perform the estimation for $\binom{n}{2}$ kernel pairs. Given that each kernel pair can have 16 possible block ratios, the total number of estimations is $\frac{16n(n-1)}{2}$. However, note that this computation is only done once in the beginning and we expect n to be quite small. Later on, as a new kernel joins the pool, we only need to make $16n$ estimations to incrementally update the energy-efficiency improvement table where n is the number of existing kernels in the waiting pool. Our experimental evaluation confirms that the overhead is indeed negligible.

Kernel Pair	**Block Ratio**	**(Core, Memory) Frequency MHz**	**GOPS/Watt Improvement**
K_1,K_2	5:3	(836, 324)	30%
K_2,K_3	4:5	(705, 400)	25%
.	.	.	.

Table 4: Energy-efficiency improvement for kernels pairs.

Once the energy-efficiency improvement table is updated in the event of a new kernel joining the pool or an existing kernel completing execution, we employ a greedy scheduling algorithm. We dispatch the concurrent kernel pair with the highest energy-efficiency improvement to the GPU. When one of the running kernels completes its execution, we remove this kernel completely from the energy-efficiency improvement table. Then we again select the most energy-efficient kernel pair from the modified table and dispatch them to the GPU. In some cases, concurrent execution of a kernel pair might perform worse than sequential execution specially if both kernels are similar in nature (compute-intensive or memory-intensive). If there exists no kernel pair with positive improvement, then we choose to execute the remaining kernels in sequential order using FIFO policy. During sequential execution of the kernels, a new kernel may join the pool. If this new kernel can pair up with an existing kernel to offer improvement in performance per watt, then we postpone the execution of the currently running kernel and instead dispatch the concurrent kernel pair.

Illustrative Example We now use a simple example to illustrate the scheduling algorithm. In this example, there are four kernels and each kernel has 100 blocks waiting to be processed. For simplicity of exposition, we assume that all the kernel blocks have the same execution time and they all arrive at the same time. We first compute the energy-efficiency improvement for each kernel pair as shown in Table 5. Note that the table does not contain all possible kernel pairs. This is because the remaining kernel pairs result in degradation of energy-efficiency through concurrent execution compared to the sequential execution.

We dispatch the kernel pairs $\{K_1, K_2\}$ first with block ratio 5:3 and core, memory frequency setting at (836 MHz, 324 MHz). For this concurrent kernel, 100 blocks of K_1 needs 60 blocks of K_2 to maintain the 5:3 block ratio. As the execution time of a block of K_1 is identical to that of a block of K_2, the kernel K_1 completes execution first because it has larger share in the concurrent kernel mix. At this point, we still have 40 blocks of $K2$ to process. If the block execution time were different for different kernels, then the number of remaining blocks might be different. This concurrent execution of $\{K_1, K_2\}$ improves the energy-efficiency by 30% compared to the sequential execution. As K_1 has completed its execution, we remove all entries with K_1 from the table. The updated table is shown in Table 6.

Table 5: Initial kernels and energy-efficiency improvement

Kernel	Available blocks
K_1	100
K_2	100
K_3	100
K_4	100

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement
K_1,K_2	5:3	(836, 324)	30%
K_2,K_3	4:5	(705, 400)	25%
K_3,K_4	3:5	(967, 710)	20%
K_1,K_4	2:6	(836, 324)	15%

Table 6: Updated information after K_1 in $\{K_1, K_2\}$ completes

Kernel	Available blocks
K_2	40
K_3	100
K_4	100

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement
K_2,K_3	4:5	(705, 400)	25%
K_3,K_4	3:5	(967, 710)	20%

Table 7: Updated information after K_2 in $\{K_2, K_3\}$ completes

Kernel	Available blocks
K_3	50
K_4	100

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement
K_3,K_4	3:5	(967, 710)	20%

Table 8: Updated information after K_3 in $\{K_3, K_4\}$ completes

Kernel	Available blocks
K_4	17

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement

Now we run concurrent kernel $\{K_2,K_3\}$ in block ratio 4:5 and core, memory frequency setting at (705 MHz, 400 MHz). In this case, 40 blocks of K_2 and 50 blocks of K_3 run concurrently to maintain the block ratio. As K_2 had only 40 blocks, it completes execution first and we are still left with 50 blocks of K_2. The improvement for this concurrent execution compared to the sequential execution is 25%. We now remove K_3 from the table and Table 7 shows the updated kernel information and efficiency improvement.

Now we will run the concurrent kernel $\{K_3,K_4\}$ at block ratio 3:5 and core, memory clock speed (967MHz, 710 MHz). In this case, 50 blocks of K_3 need 83 blocks of K_4 to maintain the block ratio. After K_3 completes execution, only 17 blocks of K_4 are left. Table 8 shows the current kernel information and efficiency improvement after K_3 completes execution.

Now there is only K_4 left. So we run it alone at its most energy-efficient frequency. This completes the execution of all the 400 blocks from the four kernels. The exact energy-efficiency improvement compared to the sequential execution depends on the number of blocks per kernel and the relative execution time of the kernels.

6. Performance-Power Estimation

In this section, we present our power-performance estimation models. Given a concurrent kernel pair and a specified block ratio, our goal is to estimate the optimal performance per watt (GOPS/Watt) by varying the frequency settings. We subsequently also need to find the optimal performance per watt for sequential execution of the kernels by choosing appropriate frequency setting for each kernel. Therefore, we first present the model to estimate the optimal frequency setting for a single kernel. A concurrent kernel pair can then be treated as a fused kernel and we will show the estimation approach for a fused kernel.

6.1 Frequency Selection for Single Kernel

This section introduces the model to estimate the frequency setting that results in optimal performance per watt for a single kernel. As mentioned before, the underlying platform provides 36 possible frequency pairs for core, memory clocks. In next-generation platforms, the number of discrete frequency levels per component can be even higher. Hence, it is not feasible to execute the kernel at each possible frequency pair setting to identify the optimal one. Instead we employ a neural network model for this problem. The input to our estimation model are the features of the kernel at the highest frequency pair. The output of the model are the optimal GOPS/Watt and the corresponding frequency setting. We first introduce the feature selection and then we present the neural network model.

6.1.1 Kernel Feature Selection

We choose the kernel features that cover the main GPU components and also impact the performance of the kernel. We select a subset of features from the NVIDIA Profiler [3]. The profiler provides a number of metrics; however, we observed that some of them have minimal impact on power and performance. After filtering out these irrelevant features, we are left with the features in Table 9.

A memory request may involve several transactions. For a coalesced memory request, it would cause less transactions and thus has higher energy efficiency. Therefore, we include the number of memory transactions in addition to the throughput information.

Feature	GPU Components
Single-precision FLOPS	Compute units
Double-precision FLOPS	
Special FLOPS	
Arithmetic unit utilization	
L1/Shared memory utilization	L1/Shared memory
Shared memory throughput GB/s	
Shared load/store per second	
Texture transactions per second	Texture Cache
L2 write/read per second	L2 Cache
L2 throughput GB/s	
Dram write/read per second	DRAM
Dram throughput GB/s	
Giga instructions issued per second (GOPS)	General information. They imply the usage of all GPU components.
Issued Load/Store instruction per second	
Global Load/Store transactions per second	

Table 9: Selected features and the corresponding GPU component

6.1.2 Neural Network

We use neural network to build an estimation model. In order to build a robust model and avoid over-fitting, we use a large number of micro-kernels and real-world kernels in the training set. We create 190 micro-kernels each stressing the compute and memory components of the GPU at varying capacity. Besides the micro-kernels, we include another 25 real-world kernels from Rodinia benchmark suite [4] and CUDA samples to the training set. In an offline process, we run each of the 215 training kernels at all 36 different frequency setting and find out the most energy-efficient frequency setting and the corresponding GOPS/Watt. We also measure the features of each training kernel at the highest frequency setting. Now with the input being the kernel's features at the highest frequency setting, and the output being the optimal frequency setting and the corresponding GOPS/Watt, we have 215 samples. We then use these samples to train the neural network. The neural network consists of two layers with the hidden layer having 21 neurons. We use Neural Network Toolbox in MATLAB to create and train the neural network.

The model needs to estimate GOPS/Watt and the corresponding core and memory frequencies. Neural networks cannot estimate a vector accurately. Therefore, we use three models in our estimation approach. The first model estimates the optimal GOPS/Watt given the features of the kernel at the highest frequency as input. This estimated optimal GOPS/Watt along with the feature vector are input to the core model and the DRAM model to generate the appropriate core and memory frequency setting, respectively. Thus these three models are actually correlated, and can be viewed as one model that outputs a vector with three elements: GOPS/Watt, core frequency, and memory frequency. Figure 3 shows the relationship among the three models.

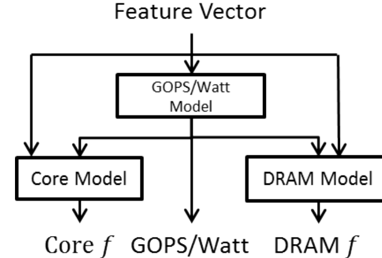

Figure 3: The relationship among the estimation models

6.1.3 Estimation Accuracy

We evaluate the accuracy of our estimation model using 28 test kernels from Rodinia benchmark suite [4] and CUDA samples. These test kernels are completely disjoint from the set of training kernels. For each test kernel, we input the feature vector at the highest frequency level to our neural network model and obtain the estimated optimal GOPS/Watt and the corresponding frequency setting. In order to obtain the actual optimal GOPS/Watt, we run each test kernel at 36 different frequency setting and identify the optimal GOPS/Watt and the frequency setting. The average error for GOPS/Watt estimation across these 28 test kernels is only 3.61%. The maximum GOPS/Watt estimation error is less than 12%.

Note that estimating the optimal frequency setting is more challenging. This is because the GPU operates at discrete frequency levels. Therefore, we round up or round down the frequency value obtained from the neural network model to the nearest discrete frequency level. Figure 4 plots the estimation accuracy of the core and memory frequency for the 28 test kernels. The X-axis is the kernel ID (1–28), while the Y-axis plots the actual and estimated optimal frequency. We estimate the core and memory frequency level correctly for 22 and 23 benchmarks, respectively. Even when the estimation does not match accurately, the estimated frequency is only one level away from the actual optimal frequency.

6.2 Performance per Watt Estimation for Concurrent Execution

So far we have shown how to estimate the optimal GOPS/Watt and the corresponding frequency setting for a single kernel given its features at the highest frequency level. Our aim now is to estimate the optimal GOPS/Watt and the corresponding frequency setting

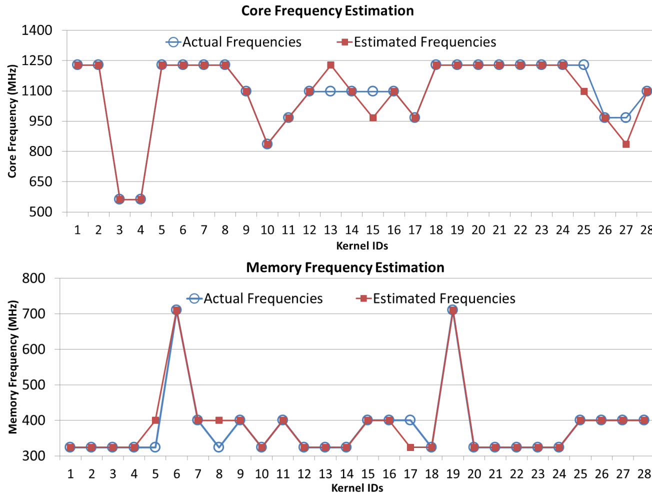

Figure 4: Accuracy of optimal frequency estimation for single kernel using neural network model.

for a concurrent kernel pair at a specified block ratio. We note that a concurrent kernel pair can be treated as a single fused kernel. Therefore, if we can estimate the feature vector of the concurrent kernel pair, we can use the same neural network model presented earlier to obtain the performance per watt and the frequency setting. In this section, we introduce the methods to estimate the features of a concurrent kernel pair and evaluate the estimation accuracy.

Let us consider a kernel pair $\{K_i, K_j\}$. Let N_i be the maximum number of blocks of kernel K_i that can be accommodated in each SMX. The value of N_i for a kernel is determined by many factors, including number of warps per block, register and memory usage per block among others. Let $n_i : n_j$ be the specified block ratio of the kernel pair K_i, K_j. In other words, we would run n_i blocks of K_i in parallel with n_j blocks of K_j.

Similar kernels Let X_i represent the feature vector of kernel K_i at the highest frequency level running with N_i blocks. Further, let $GOPS_i$ represent the GOPS feature of K_i when K_i is running by itself with maximum number of blocks N_i in each SMX. The GOPS represents the instruction issued per second; the higher the value, the more is the compute intensity of the kernel. The kernels with similar GOPS value have similar compute or memory intensity. We observe that given two kernels K_i, K_j with similar GOPS value and $n_i : n_j$ block ratio, the features of the concurrent kernel X_{ij} can be accurately estimated using the following equation.

$$X_{ij} = \frac{n_i}{N_i} \cdot X_i + \frac{n_j}{N_j} \cdot X_j \tag{1}$$

We simply take the weighted sum of the two kernels for each feature. For example, suppose two kernels K_i, K_j can each execute at most 8 blocks in parallel per SMX. If the block ratio of the two kernels is 5:3, then the features of the concurrent kernel pair can be estimated as $\frac{5}{8}X_i + \frac{3}{8}X_j$.

Kernels with different compute intensity If the kernel pair has vastly different compute intensity as captured by the GOPS feature, then we cannot use Equation 1 to estimate the feature vector of the concurrent kernel as it results in high inaccuracy. When a high compute-intensive kernel runs concurrently with a low compute-intensive kernel, we observe that the block execution time of the compute-intensive kernel becomes shorter. This is because the compute-intensive kernel now has access to more compute resources running alongside the lightweight kernel. On the other hand, the features like GOPS and various utilizations for the kernel becomes greater. Thus we add a scaling factor α_i to the weighted feature equation, as shown in Equation 2.

$$X_{ij} = \frac{n_i}{N_i} \cdot X_i \cdot \alpha_i + \frac{n_j}{N_j} \cdot X_j \cdot \alpha_j \tag{2}$$

where

$$\alpha_i = max\left\{\frac{GOPS_i}{\frac{n_i}{N_i} \cdot GOPS_i + \frac{n_j}{N_j} \cdot GOPS_j}, 1\right\}$$
$$\alpha_j = max\left\{\frac{GOPS_j}{\frac{n_i}{N_i} \cdot GOPS_i + \frac{n_j}{N_j} \cdot GOPS_j}, 1\right\}$$

The warps are the scheduling unit within an SMX. The higher the GOPS value, the more is the number of ready warps. The fraction $\frac{n_i}{N_i} \cdot GOPS_i$ estimates the number of ready warps if we only put n_i blocks of kernel K_i in each SMX. When we add n_j blocks of kernel K_j, the total number of ready warps can be estimated as $\frac{n_i}{N_i} \cdot GOPS_i + \frac{n_j}{N_j} \cdot GOPS_j$.

If this sum is smaller than $GOPS_i$, the warps of n_i blocks from K_i now have higher chances to be scheduled compared to them running with more blocks from K_i (as is the case when K_i is running by itself). Thus the n_i blocks from K_i will finish faster, and the features like utilization and bandwidth will be greater. Therefore, α_i is greater than 1 for K_i.

If $\frac{n_i}{N_i} \cdot GOPS_i + \frac{n_j}{N_j} \cdot GOPS_j$ is greater than $GOPS_i$, then we simply set $\alpha_i = 1$ as we find this to be more accurate than setting $\alpha_i < 1$. This may be caused by the improved function unit utilization for mixed operations from the concurrent kernel.

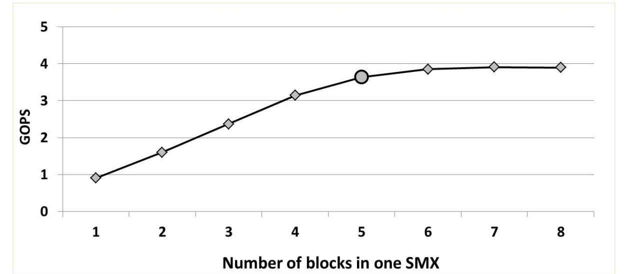

Figure 5: Finding N for kernel *Samplerank*

Memory-bound kernels For memory bound kernels, we observe that as we increase the number of blocks in SMX, the performance remains constant after a point. For example, Figure 5 plots the GOPS for the kernel *Samplerank* with increasing number of blocks in SMX. The kernel already reaches the DRAM bandwidth limit with five blocks even though the SMX can accommodate maximum eight blocks for this kernel. Thus the feature vector obtained from running eight blocks in each SMX is identical to the feature vector with five blocks per SMX. If the DRAM had unlimited memory bandwidth, the features like GOPS should have been $8/5 = 1.6$ times higher.

If we now run *Samplerank* with a compute-intensive kernel, they will be affected by the bandwidth limitation according to their

memory bandwidth requirements. We set $N_{Samplerank}$ to 5 instead of 8 to accurately capture the memory bandwidth limitation of the kernel. By changing the N value for memory-bound kernels, we can make the kernel compute-bound instead and hence can calculate the features of the concurrent kernels more accurately using scaling factor α.

As the scaling factor α can be greater than 1, while we set N to a smaller value for memory bound kernels, Equation 2 may produce a feature vector that exceeds the DRAM bandwidth limitation. Thus we add another scaling factor β.

Finally, for two kernels K_i, K_j running concurrently with block ratio $n_i : n_j$, the feature vector of the concurrent kernel at the highest frequency can be estimated using the following equation.

$$X_{ij} = \left(\frac{n_i}{N_i} \cdot X_i \cdot \alpha_i + \frac{n_j}{N_j} \cdot X_j \cdot \alpha_j \right) \cdot \beta \tag{3}$$

where

$$\beta = min \left\{ 1, \frac{MB}{\frac{n_i}{N_i} \cdot B_i + \frac{n_j}{N_j} \cdot B_j} \right\}$$

MB is the maximum memory bandwidth supported by the underlying platform and B_i is the memory bandwidth required by kernel K_i.

Estimation accuracy Given a kernel pair and a specified block ratio, we can now compute the feature vectors for the concurrent kernel at the highest frequency setting. Given these feature vectors for the concurrent kernel, we can use the neural network model to obtain the optimal GOPS/Watt for the concurrent kernel and the corresponding frequency setting. Among the 15 input features, GOPS has the highest correlation with GOPS/Watt. Thus we evaluate the feature estimation accuracy for the concurrent kernel by considering the estimation accuracy of the feature GOPS. For kernel pair with similar features, like *Matrix* and *BT*, the estimation error is the smallest. In order to stress our model, we consider the kernel pair with the largest difference in GOPS value from the benchmark kernels in Table 3, namely *Time_step* and *Bt*. Even for this kernel pair, the estimation error for GOPS is within 5%.

Finally, using the estimated features, we estimate GOPS/Watt and frequency settings of a concurrent kernel through our neural network model. For all the concurrent kernel pairs tested, the average and maximum error for GOPS/Watt estimation are 4.8% and 15% respectively. We show the relative errors between the measured and estimated GOPS/Watt of four selected kernel pairs in Figure 6 at five different block ratio.

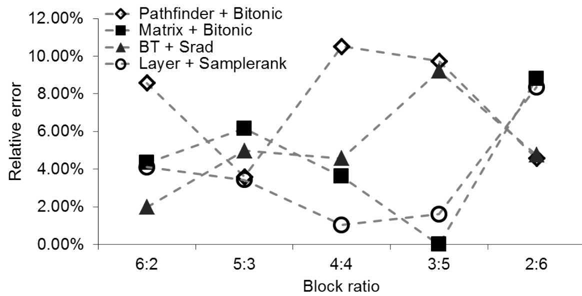

Figure 6: GOPS/Watt estimation accuracy for concurrent execution. (1) $\{$*Matrix, Bitonic*$\}$: Average error 4.7% (2) $\{$*BT, Srad*$\}$: Average error 5.1% (3) $\{$*Pathfinder, Bitonic*$\}$: Average error 7.2% (4) $\{$*Layer, Samplerank*$\}$: Average error 3.5%.

6.3 Performance per Watt Estimation for Sequential Execution

The choice of kernel pair for scheduling depends on the GOPS/Watt improvement compared to the sequential execution. Therefore, we also need to estimate the GOPS/Watt for sequential execution of a kernel pair. We use an analytical model to estimate the optimal GOPS/Watt for sequential execution of the kernel pair $\{K_i, K_j\}$.

Let us assume that the kernel pair $\{K_i, K_j\}$ when executing concurrently uses a block ratio of $n_i : n_j$. Let us further assume that during concurrent execution of the kernels, we can process $inst_i$ and $inst_j$ instructions corresponding to the kernels K_i, K_j in unit time. The values of $inst_i$ and $inst_j$ depend on the block ratio as well as the processing time for each block (which can be different for different kernels). The ratio $\frac{inst_i}{inst_j}$ can be computed as follows. N_i is the maximum number of blocks of K_i that can be accommodated per SMX. For memory-bound kernels, the computation of N_i has been presented previously. Then

$$\frac{inst_i}{inst_j} = \frac{\frac{n_i}{N_i} \cdot GOPS_i}{\frac{n_j}{N_j} \cdot GOPS_j}$$

Our goal is to compute the performance per watt for sequential execution of the two kernels where we process $inst_i$ instructions of K_i and $inst_j$ instructions of K_j running each kernel at its most energy-efficient frequency setting.

Let P_i represent the power consumption and G_i represent the GOPS of kernel K_i at its most energy-efficient frequency setting. Then the execution time of kernel K_i represented as t_i is

$$t_i = \frac{inst_i}{G_i}$$

The energy consumed during the execution of the two kernels

$$E = t_i \cdot P_i + t_j \cdot P_j$$

Finally, we can express the GOPS/Watt of the sequential execution S_{ij} as:

$$S_{ij} = \frac{inst_i + inst_j}{E}$$

After simplification, the equation becomes:

$$S_{ij} = \frac{1}{\frac{P_i \cdot inst_i}{G_i \cdot (inst_i + inst_j)} + \frac{P_j \cdot inst_j}{G_j \cdot (inst_i + inst_j)}}$$

So how can we estimate S_{ij} given the information available to us? $\frac{P_i}{G_i}$ is simply the reciprocal of the optimal GOPS/Watt for kernel K_i and can be estimated using the neural network model presented in Section 6.1.

Figure 7 shows the accuracy of the GOPS/Watt estimation for sequential execution of four selected kernel pairs. The results for the other kernel pairs are similar and are not shown due to space constraints. The results show that our estimation is generally quite accurate. For all the kernel pairs we tested, the maximum error is only 10.1%.

6.4 Putting it all together

To summarize, our framework has an offline component that builds the estimation model for a given platform using a set of microbenchmarks and real-world kernels. This model is built only once for each platform.

The online scheduler for concurrent kernel execution and DVFS requires the following offline information for each kernel: (a) the features of the kernel at the highest frequency level, which only

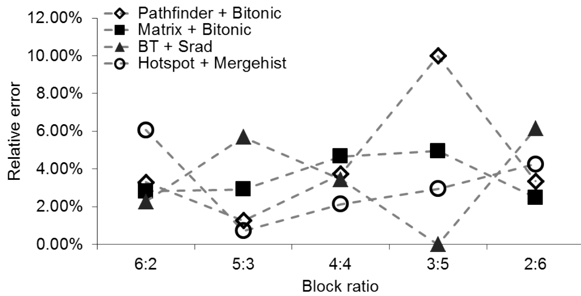

Figure 7: GOPS/Watt estimation accuracy of sequential execution. (1) {*BT, Srad*}: Max error 6.1% (2) {*Pathfinder, Bitonic*}: Max error 9.9% (3) {*Matrix, Bitonic*}: Max error 5.3% (4) {*Hotspot, Mergehist*}: Max error 6.1%.

requires one execution of the kernel, and (b) the maximum number of blocks N_i that can be supported per SMX for the kernel. It is easy to estimate N_i given the features of the kernel and does not require additional execution of the kernel.

At runtime, given the features of each kernel, the estimation model can compute the optimal GOPS/Watt for both sequential and concurrent execution of a kernel pair. The scheduler can then identify the pair that has maximum performance per watt improvement through concurrent execution, the corresponding block ratio, and the frequency setting. If none of the kernel pairs is suitable for concurrent execution, then the scheduler simply proceeds with sequential execution of the kernels in the waiting pool, each at its most energy-efficient frequency setting obtained through the estimation model.

7. Experiment Evaluation

In this section, we evaluate our approach with the experimental setup presented in Section 3.

We first study the quality of the solutions returned by our approach based on estimation compared to the actual optimal solution. In particular, given a kernel pair, our algorithm can quickly determine the optimal GOPS/Watt achievable by this kernel pair through concurrency and the corresponding block ratio and the frequency setting by using the power-performance estimation models. We also exhaustively run each kernel pair with all possible block ratio (16 different block ratio) and frequency settings (36 different frequency setting) for a total of $16 \times 36 = 576$ design points. This exhaustive search provides us with the actual optimal GOPS/Watt for the kernel pair.

Figure 8 compared our estimation based approach with the optimal one for eight different kernel pairs. The trends are the same for the remaining kernel pairs and are not shown here due to lack of space. The X-axis shows the kernel pairs while the Y-axis corresponds to the GOPS/Watt improvement with our approach and the optimal one (through exhaustive search) compared to the sequential execution. Clearly, the improvement strongly depends on the complementary nature of the kernels in a pair. More importantly, the results confirm that our estimation based approach can produce near-optimal solutions. The difference between our approach and the optimal solution is less than 5% across all kernel pairs. The advantage of our approach is the significantly reduced runtime to obtain the solution. The exhaustive search takes around a day to obtain the optimal solution, while our approach takes few microseconds to produce a solution that is quite close to the optimal one.

Notice that our algorithm selects the block ratio and the frequency setting for a kernel pair that is expected to produce the optimal GOPS/Watt. We now run the kernel pair with this block ratio and the frequency setting to obtain the actual GOPS/Watt improvement. This is the real improvement in performance per watt we observe from concurrent execution of the kernel pair. Table 10 presents the actual GOPS/Watt improvement using the estimated block ratio and frequency settings for 15 kernel pairs. As mentioned before, we use 11 benchmark kernels shown in Table 3 for this experiment. These 11 benchmarks result in 55 different kernel pairs. Table 10 shows the results for the top 15 kernel pairs that improve the performance per watt significantly through concurrency. As can be seen from Table 10, all the top energy-efficient concurrent kernel pairs are composed of one compute-intensive kernel and one memory-intensive kernel. Moreover, the block ratio and frequency setting vary for different kernel pairs. This confirms that our estimation models are required to tune the runtime kernel settings to achieve high performance.

So far, we have discussed performance per watt improvement for each kernel pair. If there are only two kernels in the waiting pool, the improvement can be significant. For example, the kernel pair {*Hotspot, Mergehist*} can achieve 34.5% improvement through concurrency compared to the default execution. We then put all the 11 benchmark kernels from Table 3 in the waiting pool with the given input size (number of blocks) and let our concurrent kernel scheduling algorithm dispatch the kernels in pairs to obtain the best energy efficiency. The overall GOPS/Watt improvement in this case is 20.3% by using our scheduling algorithm compared to the default execution.

Kernel Pair	**Block Ratio**	**(Core, Memory) Frequency MHz**	**GOPS/Watt improvement**
Hotspot, Mergehist	2:6	(1228, 324)	34.5%
Pathfinder, Bitonic	3:5	(1097, 324)	29.8%
Samplerank, Hotspot	4:4	(1228, 625)	28.5%
Samplerank, BT	5:3	(562, 400)	28.4%
Mergehist, Matrix	7:1	(1228, 324)	26.0%
Hotspot, Transpose	4:4	(1228, 400)	23.4%
Matrix, Bitonic	3:5	(836, 324)	23.2%
Layer, Time_step	4:4	(1228, 480)	23.1%
Layer, Bitonic	5:3	(1228, 400)	22.6%
Hotspot, Time_step	4:5	(967, 400)	21.6%
Hotspot, Srad	5:3	(1097, 400)	20.2%
Hotspot, Matrix	4:4	(1228, 400)	18.6%
Bt, Srad	3:5	(967, 480)	16.0%
Layer, Pathfinder	2:6	(1097, 400)	14.3%
Matrix, transpose	7:1	(1228, 400)	13.8%

Table 10: 15 kernel pairs with most performance per watt improvement through concurrency

We now show that the performance per watt improvement due to our approach does not come at the cost of degraded performance.

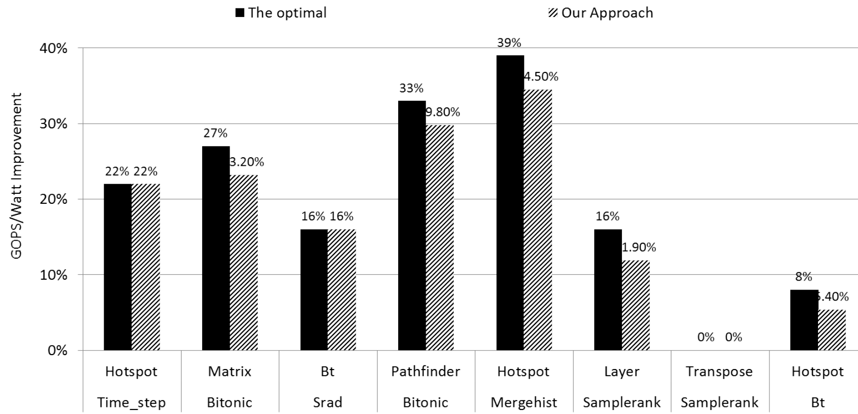

Figure 8: Comparison of our approach based on estimation and the optimal solution for different kernel pairs.

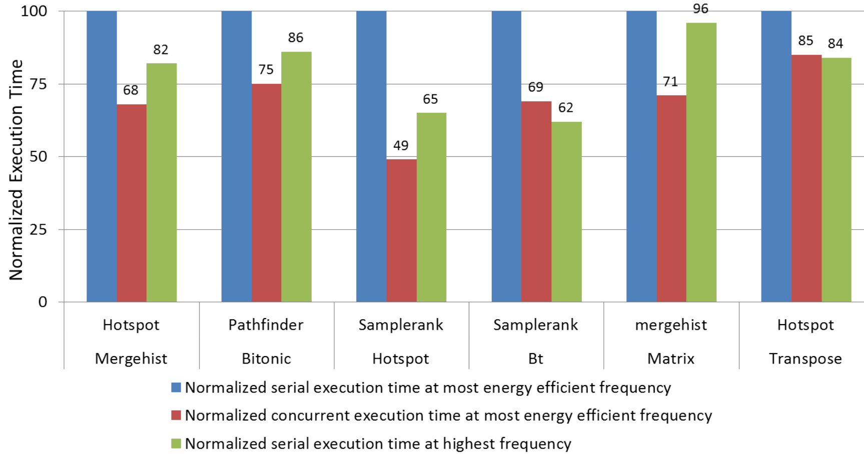

Figure 9: Performance impact of energy-efficient concurrent execution compared to sequential execution.

Indeed, in most cases, concurrent execution improves throughput compared to sequential execution in addition to energy efficiency. To validate this claim, we choose the top 6 kernel pairs from Table 10. These kernel pairs enjoy maximum benefit from concurrency. We then run each of these kernel pairs in three different ways: (a) concurrent execution to obtain optimal performance per watt, (b) sequential execution to obtain optimal performance per watt, and (c) serial execution at the highest frequency setting. Figure 9 plots the execution time corresponding to these three approaches. Clearly, concurrent execution improves the performance over energy-efficient sequential execution substantially. This is mainly due to the better utilization of the resources. More importantly, concurrency even manages to improve the execution time compared to the sequential execution at the highest frequency setting for 4 out of 6 kernel pairs. For the remaining kernel pairs, the performance loss is minimal, while the performance per watt improvement is significant to justify concurrency.

Finally, we evaluate the possibility of running more than two kernels concurrently. Our approach can be easily extended to more than two kernels. In particular, we consider concurrent execution of three kernels. Based on Table 10, we observe that there are five groups of three kernels that have the highest chance of improving energy efficiency. For each kernel group, we exhaustively search all block ratios and frequencies to find out the optimal GOPS/Watt improvement. We plot the results in Figure 10. We also plot the improvement when running two kernels from the group concurrently and leaving one kernel to execute in isolation. As can be seen, the concurrent execution with three kernels does not produce higher energy efficiency for any of the kernel groups. Indeed running two kernels concurrently has higher GOPS/Watt improvement. The reason may be explained as follows. Although a concurrent combination of three kernels may have more balanced utilization of the compute units and the memory bandwidth, as the power consumption of the cores and the memory can be reduced by frequency scaling, a more compute- or memory-intensive kernel could have higher energy efficiency than this concurrent kernel.

8. Related Work

Recently, using concurrent kernel execution to improve the GPGPU performance has been studied in many papers. Software solutions are proposed to enable running multiple GPU kernels concurrently. Guevara et al. [9] study the concurrent kernel execution through

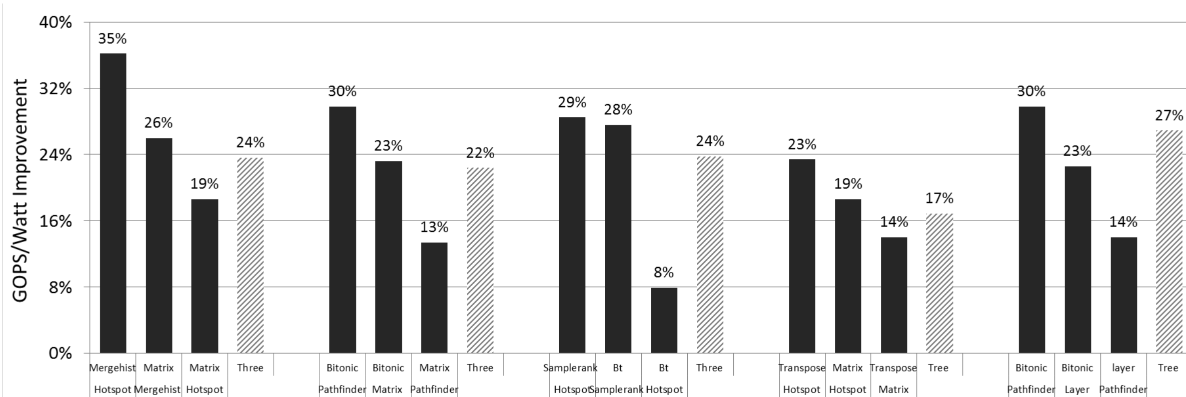

Figure 10: Performance per watt improvement with three concurrent kernels compared to two concurrent kernels

source code transformation. Two kernels are combined into a single kernel function using thread interleaving. Wang et al. [15] propose three methods to run kernels concurrently, which are inner threads, inner thread blocks and inter thread blocks. Furthermore, Gregg et al. [8] propose a technique that is similar to thread interleaving to merge kernels for fine-grained resource sharing.

With hardware support for concurrent kernel execution, more studies are conducted on modern generation GPUs. Pai et al. [14] identify that the left over policy is the main reason of inefficiency when running kernels concurrently on NVIDIA Fermi GPUs. They propose elastic kernels to address this issue. Adriaens et al. [5] propose to spatially partition GPU resource for concurrent execution. Zhong et al. [16] propose an algorithm to combine two suitable kernels with optimized slicing length to improve the GPU throughput. More recently, Liang et al. [12] utilize concurrent kernel execution to achieve GPU spatial-temporal multitasking.

However, none of the previous studies considers using the concurrent kernel execution technique to improve energy efficiency. Instead, we explore using this technique to improve the performance as well as achieve better energy efficiency.

9. Conclusion

We improve GPU energy-efficiency for computational kernels through a combination of concurrent execution and DVFS. To exploit concurrency, we need to determine the appropriate subset of kernels to be executed together, the proportion of each kernel in the mix, and the energy-efficient frequency settings. We develop a set of analytical and neural network based models to estimate the power-performance behavior of concurrent and sequential execution of kernel pairs accurately and efficiently. The concurrent execution of kernel pairs can improve performance per watt by upto 34.5% with either improvement or minimal degradation of throughput. Given a set of eleven kernels, our concurrent scheduling algorithm in combination with the estimation models improve performance per watt by 20.3% compared to sequential execution.

Acknowledgments

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2012-T2-1-115.

References

[1] AMD Radeon^{TM} R9. http://www.amd.com/en-us/products/graphics/desktop/r9/295x2.

[2] NVidia GTX TITAN Z GPU. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-z/specifications.

[3] Profiler User's Guide. http://docs.nvidia.com/cuda/profiler-users-guide.

[4] Rodinia Benchmark. http://www.cs.virginia.edu/ skadron/wiki/rodinia/index.php.

[5] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *International Symposium on High Performance Computer Architecture*, 2012.

[6] J. Chen. Gpu technology trends and future requirements. In *IEEE International Electron Devices Meeting*, 2009.

[7] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[8] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, 2012.

[9] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, 2009.

[10] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the gpu. In *International Conference on Green Computing and Communications*, 2010.

[11] J. Lee, V. Sathisha, M. Schulte, K. Compton, and N. S. Kim. Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling. In *International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[12] Y. Liang, H. Huynh, K. Rupnow, R. Goh, and D. Chen. Efficient gpu spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[13] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 1965.

[14] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[15] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *International Conference on Green Computing and Communications*, 2010.

[16] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2014.