Dynamic GPGPU Power Management Using Adaptive Model Predictive Control

Abhinandan Majumdar* Leonardo Piga† Indrani Paul† Joseph L. Greathouse† Wei Huang† David H. Albonesi*
*Computer System Laboratory, Cornell University, Ithaca, NY, USA
†AMD Research, Advanced Micro Devices, Inc., Austin, TX, USA
*{am2352, dha7}@cornell.edu, †{Leonardo.Piga, Indrani.Paul, Joseph.Greathouse, WeiN.Huang}@amd.com

***Abstract*—Modern processors can greatly increase energy efficiency through techniques such as dynamic voltage and frequency scaling. Traditional predictive schemes are limited in their effectiveness by their inability to plan for the performance and energy characteristics of upcoming phases. To date, there has been little research exploring more proactive techniques that account for expected future behavior when making decisions.**

This paper proposes using Model Predictive Control (MPC) to attempt to maximize the energy efficiency of GPU kernels without compromising performance. We develop performance and power prediction models for a recent CPU-GPU heterogeneous processor. Our system then dynamically adjusts hardware states based on recent execution history, the pattern of upcoming kernels, and the predicted behavior of those kernels. We also dynamically trade off the performance overhead and the effectiveness of MPC in finding the best configuration by adapting the horizon length at runtime. Our MPC technique limits performance loss by proactively spending energy on the kernel iterations that will gain the most performance from that energy. This energy can then be recovered in future iterations that are less performance sensitive. Our scheme also avoids wasting energy on low-throughput phases when it foresees future high-throughput kernels that could better use that energy.

Compared to state-of-the-practice schemes, our approach achieves 24.8% energy savings with a performance loss (including MPC overheads) of 1.8%. Compared to state-of-the-art history-based schemes, our approach achieves 6.6% chip-wide energy savings while simultaneously improving performance by 9.6%.

I. Introduction

Dynamic voltage and frequency scaling (DVFS) is a power-saving mechanism that places devices such as CPUs, GPUs, and DRAM channels into lower performance states in order to save power. By using low-power states when they will not greatly affect performance, significant energy can be saved without slowing down the application. Good DVFS policies are vital, since poor decisions can cause both performance and energy losses.

Existing DVFS-based power management techniques, such as AMD's Turbo Core [1], [2] and Intel's Turbo Boost [3], [4], [5], select performance states based on the chip activity seen in the recent past. This may lead to performance and efficiency losses, since this fails to anticipate future performance demands. For instance, lowering the frequency for the next time step may reduce power at the cost of lost performance, while the same action at a future time step may save the same power with no performance loss. Both techniques may equally reduce power, but the latter will yield better energy and performance. This work attempts to alleviate this problem in general-purpose GPU (GPGPU) compute-offload applications.

Previous work that statically optimized individual GPGPU kernels [6], [7], or dynamically optimized over multiple iterations of each kernel [8], [9], [10], ignoring future kernel behavior; they utilize information from the last timestep to predict hardware configurations for the next. This falls short for applications with multiple interleaved kernels with different characteristics and for irregular applications with kernels that vary across iterations [11]. Moreover, these approaches treat each kernel equally in terms of power management decisions, even though kernels may widely vary in their impact on overall application performance. As a result, they may not be able to "catch up" for lost performance or energy savings in later phases with unanticipated behavior. CPU schemes such as phase tracking/prediction only consider the performance of the immediate phase. Similarly, Chen et al. [12] predict the performance of the immediate phase but ignore past behavior.

This paper presents a GPGPU power management approach that performs inter-kernel optimization while accounting for future kernel behavior. The approach anticipates the expected pattern of future kernels, and their performance and power characteristics, in order to optimize overall application performance and energy. A key component of our approach is *model predictive control (MPC)*. MPC optimizes for a future prediction horizon in a receding manner but applies the optimal configuration at the current timestep. However, the implementation overheads of a full MPC algorithm make it unsuitable for the timescales of chip-level dynamic power management, as the problem of maximizing kernel-level energy efficiency under a given performance target is NP-hard. We propose new greedy and heuristic approximations of MPC that are effective at saving energy with modest performance loss yet applicable to runtime power management. Furthermore, we dynamically adjust the prediction horizon in order to limit the performance overhead caused by MPC.

To determine the appropriate hardware configuration for a kernel, we develop a prediction model to estimate kernel-level performance and power at different hardware configurations and a pattern extractor that predicts which kernels will appear in the future. Our overall approach permits MPC to proactively limit performance losses by dynamically expending more energy on high-throughput kernels. MPC also avoids spending a disproportionate amount of energy on low-throughput kernels. Instead, it seeks opportunities from the future high-throughput phases to compensate for the performance lost when low-throughput kernels are run at slow DVFS states.

Our approach saves 24.8% energy with a performance loss of 1.8% compared to AMD Turbo Core and reduces energy by 6.6% while improving performance by 9.6% with respect to state-of-the-art history-based power management schemes.

TABLE I: Software visible CPU, Northbridge, and GPU DVFS states on the AMD A10-7850K.

CPU P States	Voltage (V)	Freq (GHz)
P1	1.325	3.9
P2	1.3125	3.8
P3	1.2625	3.7
P4	1.225	3.5
P5	1.0625	3.0
P6	0.975	2.4
P7	0.8875	1.7

NB P States	Freq (GHz)	Memory Freq (MHz)
NB0	1.8	800
NB1	1.6	800
NB2	1.4	800
NB3	1.1	333

GPU P States	Voltage (V)	Freq (MHz)
DPM0	0.95	351
DPM1	1.05	450
DPM2	1.125	553
DPM3	1.1875	654
DPM4	1.225	720

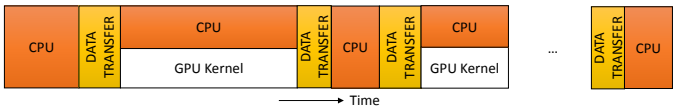


Fig. 1: Typical GPGPU application phases.

II. Background and Motivation

A. Heterogeneous Processor Architectures

Modern heterogeneous processors consist of CPU cores integrated on the same die with GPU cores and components such as a northbridge (NB) and power controllers. Power and thermal budgets may be shared across resources, and some devices (e.g., the GPU and NB) may share voltage rails.

Table I shows the different DVFS states for the CPU, NB, and GPU in the AMD A10-7850K processor that we study in this work. Changing NB DVFS impacts memory bandwidth, since each state maps to a specific memory bus frequency. All CPU cores share a power plane. The GPU is on a separate power plane, which it shares with the NB; the NB and GPU frequencies can be set independently, but they share a common voltage.

Lower CPU DVFS states reduce the CPU power and can slightly reduce the GPU power due to a reduction in temperature and leakage. GPU DVFS states change the core frequency of the GPU; however, higher NB states can prevent reducing the GPU's voltage along with the frequency. This can limit the amount of power saved when changing GPU DVFS states. Similarly, if the GPU is at a high power state, reducing the NB state may only change the NB frequency.

B. GPGPU Program Phases

A breakdown of a typical GPGPU application is shown in Figure 1. The host CPUs first perform some amount of work, shown as *CPU*. After this, they launch computational kernels to the GPU. A kernel consists of parallel workgroups that are comprised of parallel threads. While the GPU is busy doing computation, the CPUs may be waiting for the GPU to finish, preparing data for launching the next GPU kernel, or running parts of the computation concurrently with the GPU.

The relative amount of time spent in each phase varies across applications and inputs. For the workloads we investigate in this paper, the CPU and GPU have little overlapping execution. We therefore focus on the power efficiency and performance of the GPU kernel execution phases and leave workloads that simultaneously exercise the CPUs and GPU or concurrent GPU kernels as future work [13].

TABLE II: Execution pattern of three irregular benchmarks. Here, A^i indicates kernel A repeats i times. F_1 to F_9 are invocations of the same kernel F, each taking different inputs.

Benchmark	Kernel Execution Pattern
Spmv	$A^{10}B^{10}C^{10}$
kmeans	AB^{20}
hybridsort	$ABCDEF_1F_2F_3F_4F_5F_6F_7F_8F_9G$

C. GPGPU Kernel Characterization

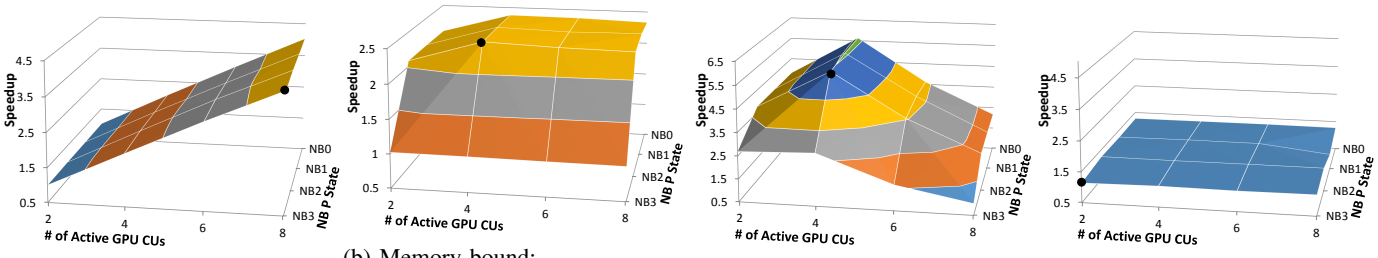
GPGPU kernels show sensitivity to hardware configurations and a range of performance and power scaling behavior. Figure 2 shows the relative performance of example GPU kernels as NB DVFS states and the number of active GPU compute units (CUs) are varied. Each graph contains a mark at the energy-optimal point.

These kernels reach their best efficiency at different configurations. Compute-bound kernels perform better with more CUs, and their energy-optimal point is at a lower NB state. Memory-bound kernels are sensitive to higher NB states, but the performance saturates from NB2 onwards because NB2 through NB0 have the same DRAM bandwidth. Peak kernels maximize performance and minimize energy at a lower hardware configuration due to destructive shared cache interference [14], [15], [16]. Finally, the performance of unscalable kernels is not affected by hardware changes; these achieve high energy efficiency at the lowest GPU configuration. These results demonstrate that mischaracterization can lead to sub-optimal performance or energy.

D. Kernel Runtime Execution Diversity

Table II shows the execution pattern of the kernels of three benchmarks represented using regular expression. *Spmv*, from a modified version of SHOC [17], runs three sparse matrix vector multiplication algorithms ten times each. The *kmeans* application from Rodinia [18] runs the *swap* kernel once, and then iterates the *kmeans* kernel 20 times. The *hybridsort* application from Rodinia runs six different kernels, with the kernel *mergeSortPass* iterating nine times, each with different input arguments. Each kernel achieves energy optimality at different hardware configurations.

Figure 3 shows how the kernel instruction throughput (normalized to the overall throughput) varies during an application's execution. We observe that *Spmv* transitions from high- to low-throughput phases, while *kmeans* demonstrates a low- to high-throughput transition. *Hybridsort* shows multiple phase



(a) Compute-bound: *MaxFlops* (b) Memory-bound: *readGlobalMemoryCoalsced* (c) Peak: *writeCandidates* (d) Unscalable: *astar*

Fig. 2: Performance trends and energy-optimal points of GPGPU kernels at different hardware configurations.

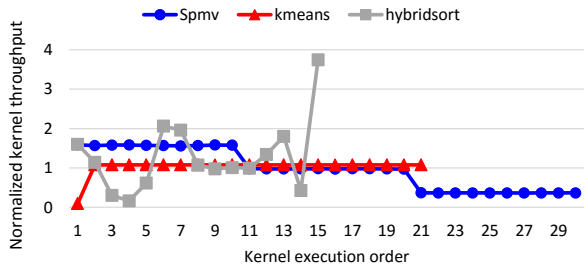


Fig. 3: Kernel throughput for *Spmv*, *kmeans* and *hybridsort*. The y-axis is normalized to the overall throughput.

transitions not only among kernels, but even by the same kernel taking different input arguments. These characteristics are typically seen in irregular applications. For example, graph algorithms can vary across input and iteration [11].

E. Potential of "Future-Based" Schemes

Our goal is to minimize energy while meeting a performance target, in our case, the performance of AMD Turbo Core, which we describe in Section V-B. In this section, we perform a limit study using two configuration decision algorithms. Both approaches have perfect knowledge of the effect of every hardware configuration on kernel performance and power. The latter also knows the exact pattern of future kernel executions, as well as their performance and power characteristics. Thus, these results could not be obtained in a real system with imperfect predictions.

The *Predict Previous Kernel* (PPK) algorithm attempts to minimize energy while assuming the previous kernel will repeat next. It does not look further in the future, but makes its decision based on perfect knowledge of the performance and power characteristics of every hardware configuration with respect to the just completed kernel. PPK represents a best-case scenario for current state-of-the-art history-based algorithms [8], [9], [19], which in practice have errors in their performance and power predictions. In contrast, the *Theoretically Optimal* (TO) algorithm performs a full state space exploration of all future kernels and finds the globally optimal hardware configuration for each kernel iteration.

Figure 4 compares the energy and performance of these algorithms against Turbo Core on the AMD A10-7850K. We observe that PPK matches TO for regular benchmarks such as *mandelbulbGPU*, *NBody* and *lbm*. These benchmarks have a single kernel iterating multiple times; thus, future

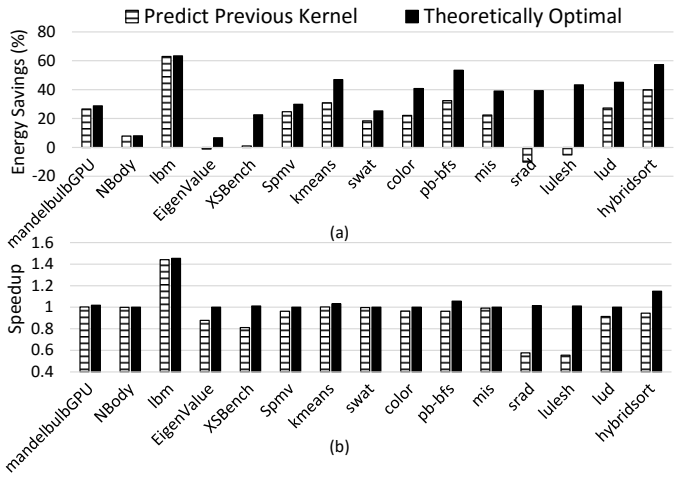


Fig. 4: Comparison of Predict Previous Kernel and Theoretically Optimal algorithms. (a) indicates energy savings and (b) speedup over AMD Turbo Core.

knowledge is not helpful. However, for the remaining irregular benchmarks, PPK consumes more energy (up to 48%) and degrades performance (up to 46%) compared to TO.

To understand why future knowledge can be so beneficial, we reconsider the benchmarks *hybridsort*, *Spmv*, and *kmeans*, shown in Figure 3. In *hybridsort*, all of the kernel invocations differ in throughput, with some varying with inputs. As a result, PPK always mispredicts the next kernel behavior, which leads to far-from-optimal performance and energy results. The applications *XSBench*, *srad* and *lulesh* exhibit similar behavior.

Spmv, on the other hand, exhibits two high-to-low throughput transitions. While this behavior results in only two mispredictions by PPK, the performance loss compared to the baseline is 4%. This is because PPK reduces the performance of the initial high-throughput phase in order to save energy. On encountering future low-throughput phases, PPK is unable to increase the performance enough to make up for the lost performance; even the highest-powered hardware configuration does not suffice. As such, PPK suffers a performance loss with respect to Turbo Core. The application *lud* shows a similar high-to-low throughput transition. This result demonstrates the benefits of not only anticipating future kernel patterns, but the performance characteristics of these future kernels as well.

In contrast to *Spmv*, *kmeans* shows a single low-to-high transition. On encountering the first dominating low-throughput

kernel, PPK is temporarily unable to reach the performance target. The performance is degraded so severely that it cannot be made up in the remaining kernels even when they are run in highest power configuration, thereby consuming more energy. Unaware of the fact that future high-throughput kernels will compensate for the initial low performance, PPK achieves lower energy savings compared to the optimal algorithm. The benchmark *pb-bfs* also exhibits similar results. The *hybridsort* application has multiple high-to-low changes, and thus suffers both reduced energy savings and performance losses.

Motivated by this fundamental limitation of algorithms that ignore the future, like Predict Previous Kernel, and by the potential demonstrated by Theoretically Optimal by perfectly predicting future kernels, we propose a future-aware dynamic kernel-level power management policy. This proposed policy anticipates future kernel performance and proactively assigns hardware resources in order to meet its performance and energy targets. We show that a power management policy driven by the principle of feedback and MPC limits the performance loss while significantly improving energy efficiency.

III. Problem Formulation

In this section, we mathematically formulate the problem. The overall objective is to minimize the total kernel-level energy consumption of a GPGPU application without impacting the net kernel performance compared to AMD Turbo Core. In order to compare the performance of a given application over different hardware configurations, we adopt kernel instruction throughput as our performance metric. Equation 1 presents the formulation.

$$\begin{aligned} &\min_{\overrightarrow{s}} \sum_{i=1}^{N} E_i(s_i) \\ &\text{such that} \\ &\frac{\sum_{i=1}^{N} I_i}{\sum_{i=1}^{N} T_i(s_i)} \geq \frac{I_{total}}{T_{total}} \\ &\text{where } s_i \in S \text{ and } S = \overrightarrow{cpu} \times \overrightarrow{nb} \times \overrightarrow{gpu} \times \overrightarrow{cu} \end{aligned} \tag{1}$$

The objective is to minimize the total kernel-level application energy (E) across all N kernels while at least matching the performance of the default Turbo Core algorithm. In Equation 1, N is the total number of kernels in an application; and vectors $\overrightarrow{cpu}, \overrightarrow{nb}$, and $\overrightarrow{gpu}$ represent the CPU, NB, and GPU DVFS states, while $\overrightarrow{cu}$ represents the different ways that the GPU CUs can be activated. S is the Cartesian product of $\overrightarrow{cpu}, \overrightarrow{nb}, \overrightarrow{gpu}$ and $\overrightarrow{cu}$. The vector $\overrightarrow{s}$, which belongs to the set S, corresponds to the hardware configurations of N kernels. Each vector element s_i of $\overrightarrow{s}$ represents the hardware configuration for an i^{th} kernel. I_i and T_i are the total number of instructions (thread-count $\times$ instruction-count per thread) and execution time of the i^{th} kernel; E_i is the energy consumed by kernel i; and I_{total} and T_{total} are the total number of instructions and the execution time of all kernels in the application in the default Turbo Core approach.

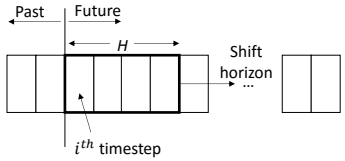


Fig. 5: Overview of the MPC process.

The theoretically optimal (TO) approach assigns a hardware configuration for each kernel instance such that the total kernel energy is minimum with no performance loss. For M possible hardware configurations and N kernels, TO requires $O(M^N)$ searches. Discrete DVFS states and GPU CUs make this problem NP-hard and thus impractical to use at runtime.[1]

Rather than exhaustive search, current runtime power management approaches optimize the next kernel in execution order based on past knowledge. To reflect this more tractable and runtime feasible approach, we reformulate Equation 1 as Equation 2.

$$\begin{aligned} &\min_{s_i \in S} E_i(s_i) \\ &\text{such that} \\ &\frac{\sum_{j=1}^{i} I_j}{\sum_{j=1}^{i} T_j(s_j)} \geq \frac{I_{total}}{T_{total}} \quad \forall 1 \leq i \leq N \text{ and } \forall s_j \in S \end{aligned} \tag{2}$$

Here for every i^{th} kernel, the optimization algorithm chooses the hardware configuration that minimizes the energy of that kernel while ensuring that the total kernel throughput thus far (including this kernel) at least matches that of the default configuration. The polynomial time complexity of $O(M \times N)$ makes the optimization tractable.

The Predict Previous Kernel (PPK) approach described earlier assumes that the last seen kernel or phase repeats again and uses its behavior to estimate the energy optimal configuration of the upcoming kernel. As shown earlier, this approach is far from optimal, which motivates our future-aware MPC approach.

IV. MPC-based Power Management

Model predictive control (MPC) is an advanced process control technique popular in various application domains [21], [22], [23], [24]. It uses a dynamic process model to proactively optimize for the current timestep by anticipating future events. An overview of the MPC process is shown in Figure 5. At each timestep i, MPC optimizes for a future horizon of H timesteps. By doing so, it captures the future events that may affect the optimal operation of the i^{th} timestep. After running the optimization, MPC applies the decision to the current i^{th} timestep. Then, for the next $(i+1)^{th}$ timestep, the horizon shifts one timestep and the algorithm optimizes over the next H timesteps. A larger H requires more computation overhead but leads to a better solution. While MPC with imperfect

[1] Formally, this can be proven by reducing the 0-1 knapsack problem, which is NP-hard [20], to finding a kernel-level energy optimal configuration without any performance loss. The formal proof is beyond the scope of this paper.

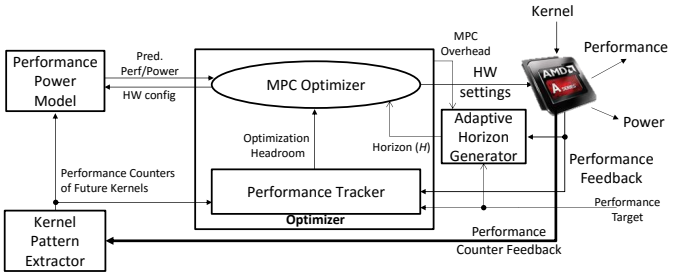


Fig. 6: MPC-based power management system.

prediction models does not guarantee global optimality, continuous feedback and proactive optimization can compensate for prediction model inaccuracies.

A. MPC-Based Online Power Management

Figure 6 shows our proposed MPC-based runtime system that attempts to minimize total energy across all kernels while avoiding performance loss. The architecture has four key components: (1) the optimizer, (2) the kernel pattern extractor, (3) the power and performance predictor, and (4) the adaptive prediction horizon generator. This framework runs as a CPU-based software policy between successive GPU kernels.

1) Optimizer: In theory, MPC minimizes energy while at least meeting the target performance. In our case, we target the performance of the default power manager. The optimizer runs the MPC algorithm to determine the per-kernel energy optimal hardware configurations (number of GPU CUs; CPU, GPU, and NB DVFS states) while maintaining the desired performance. It also tracks the past performance and instruction counts to determine the available execution time headroom. This mechanism takes as input estimates from the power and performance model, which we describe later.

a) Model Predictive Control: At each i^{th} step (kernel invocation, in this case), the MPC algorithm optimizes across a window of the next H_i kernels. It determines the minimum energy configuration across those H_i kernels that meets the ongoing performance target and uses that configuration for the current (i^{th}) kernel. After the execution of that kernel, the prediction window is shifted one kernel in time and the process is repeated for the new window of H_{i+1} kernels. The performance tracker takes the past performance as feedback and dynamically adjusts the execution time headroom for the next optimization. Equation 3 shows the MPC formulation for optimizing kernel energy across H_i future kernels under a performance target for an i^{th} kernel.

$$\min_{\vec{s}} \sum_{j=i}^{i+H_i-1} E_j(s_j)$$
$$\text{such that} \qquad (3)$$
$$\frac{\sum_{j=1}^{i+H_i-1} I_j}{\sum_{j=1}^{i+H_i-1} T_j(s_j)} \geq \frac{I_{total}}{T_{total}} \qquad \forall 1 \leq i \leq N \text{ and } \forall s_j \in S$$

MPC Search Heuristic: Traditional MPC approaches use computationally expensive backtracking [25], [26], [27] for each timestep, which is infeasible given the timescales of dynamic power management. While truly optimizing over multiple H kernels may require backtracking and involves $O\left(N \times (|\overrightarrow{cpu}| \times |\overrightarrow{nb}| \times |\overrightarrow{gpu}| \times |\overrightarrow{cu}|)^H\right)$ searches, we employ greedy and heuristic approximations that permit a polynomial time complexity of $O\left(N \times (|\overrightarrow{cpu}| + |\overrightarrow{nb}| + |\overrightarrow{gpu}| + |\overrightarrow{cu}|) \times H\right)$ to approximate the benefits of backtracking.

Our approach gathers per-kernel performance information during the first invocation of a GPGPU program in order to minimize the energy of future invocations. Using this information, it determines a search order to optimize the future kernels such that none of the optimized kernels are revisited, thereby reducing the complexity from exponential to polynomial.

The optimization algorithm attempts, in polynomial time, to address two shortcoming of previous approaches such as PPK:

1) The inability to foresee future lower-throughput kernels, which may reduce performance due to the inability to "catch up" performance-wise for aggressively saving energy in earlier, high-throughput, kernels; and
2) The inability to foresee future higher-throughput kernels, which may reduce energy savings due to the inability to compensate for overly aggressive performance settings in earlier, low-throughput, kernels.

At the conclusion of the execution of each kernel, our approach notes whether the accumulated application throughput is above the overall target throughput. Those kernels for which the overall throughput is above the target are grouped into the *above-target* cluster and those remaining grouped as *below-target*. The former group are ordered in increasing order by individual kernel performance, and then the latter group in decreasing order. The union of these two groups forms the search order for the heuristic optimization.

Figure 7 shows an example execution of a hypothetical irregular application, with the individual kernel (squares) and accumulated application throughput (solid line) normalized to the overall target throughput. The first three kernels $(1, 2, 3)$ are placed in the above-target group because their accumulated runtime throughput values (solid line) are above the overall target throughput (dashed line), while the remaining $(4, 5, 6)$ are placed in the below-target group. We order the above-target group in increasing throughput order (squares). Hence, the order is $(3, 2, 1)$. The below-target group is ordered in decreasing order; therefore the order is $(6, 5, 4)$. The overall search order becomes $(3, 2, 1, 6, 5, 4)$.

With this order determined, the next time the application is invoked, execution proceeds as follows:
Kernel 1: The optimization is performed in the order $(3, 2, 1)$. The algorithm first estimates the lowest energy configuration for kernel 3 that at least meets the overall target throughput. Any excess performance headroom carries over to kernel 2, for which the lowest energy configuration is found that meets the new target. Any accumulated excess performance headroom carries over to kernel 1, for which the lowest energy configuration is estimated again. The algorithm anticipates the

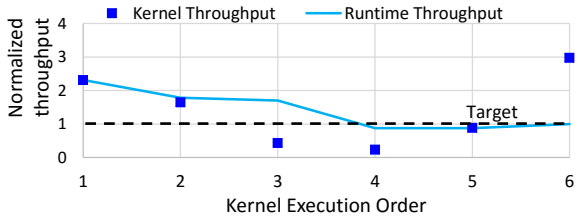


Fig. 7: An example showing the kernel throughout (squares) and overall application throughput (solid line) during the execution of a hypothetical irregular application. The *y* axis is normalized to the overall target throughput.

future drop in throughput, which guards against aggressively reducing kernel 1 performance such that it cannot be "made up" in future low performance kernels 2 and 3.

Kernel 2: The optimization order is $(3, 2)$. The algorithm first finds the lowest energy configuration for kernel 3 that at least meets the overall target throughput, taking into account the overall performance thus far. Any excess performance is carried over to kernel 2, for which the lowest energy configuration is found.

Kernel 3: The optimization order is (3); that is, the optimization only considers the current kernel.

Kernel 4: The optimization order is $(6, 5, 4)$. Since the first three kernels have already executed, they are no longer considered. At this point, the algorithm attempts to foresee future higher-throughput kernels (6 and 5) in order to trade off performance for increased energy savings for kernel 4.

Kernels 5 and 6 are optimized in a similar manner.

Greedy Hill Climbing Optimization: To reduce the search overhead and avoid an exhaustive exploration of all possible hardware configurations, we employ greedy hill climbing.

Among the hardware knobs, i.e., the DVFS states (CPU, NB or GPU) and GPU CUs, the algorithm first estimates their energy sensitivities[2] using the prediction model, and sorts them in descending order. The knob with the highest energy sensitivity is selected first and then its corresponding configuration is searched in a hill-climbing fashion such that the predicted kernel energy continues to decrease while meeting the default performance target. The search stops once the energy increases. The optimization then continues with the next highest energy sensitive knob, and so on. In the event that the algorithm fails to meet the overall performance requirements, it defaults to an empirically determined fail-safe configuration of [P7, NB2, DPM4, 8 CUs].

While this approach compromises optimality, the number of energy evaluations reduces from $(|\overrightarrow{cpu}| \times |\overrightarrow{nb}| \times |\overrightarrow{gpu}| \times |\overrightarrow{cu}|)$ to $(|\overrightarrow{cpu}| + |\overrightarrow{nb}| + |\overrightarrow{gpu}| + |\overrightarrow{cu}|)$, or a factor of 19×. The greedy search in conjunction with the MPC heuristic reduces the search cost by $65\times$ compared to an exhaustive MPC search involving backtracking, which makes our approach suitable for runtime optimization.

b) Performance Tracker: The performance tracker dynamically adjusts the execution time headroom for MPC optimization based on the desired performance target, execution history of past kernels, and performance behavior of future kernels. The performance requirement for an i^{th} kernel is enforced according to Equation 4.

$$\frac{\sum_{j=1}^{i-1} I_j + \mathbb{E}[I_i]}{\sum_{j=1}^{i-1} T_j(s_j) + \mathbb{E}[T_i(s_i)]} \geq \frac{I_{total}}{T_{total}} \tag{4}$$

The headroom for MPC optimization is dynamically adjusted using the net performance of the past $i-1$ kernels and performance counters from the kernel pattern extractor. The expected kernel time $\mathbb{E}[T_i]$, provided by the performance predictor, must be within this updated headroom (Equation 5). Significant performance slack provides the optimizer with the opportunity to aggressively save energy. With less headroom, the optimizer operates more conservatively, choosing higher performance, and higher energy, configurations.

$$\mathbb{E}[T_i(s_i)] \leq \left(\sum_{j=1}^{i-1} I_j + \mathbb{E}[I_i]\right) \Big/ \left(\frac{I_{total}}{T_{total}}\right) - \sum_{j=1}^{i-1} T_j(s_j) \tag{5}$$

2) Kernel Pattern Extractor: GPGPU applications commonly execute many kernels in a regular order. As shown in Section II-D, several applications present regular execution patterns. There may also be distinct patterns within the same kernel across multiple invocations due to input data set changes. We use these patterns to predict the future behavior of the kernels and to store their performance counters for future use by the optimizer. The mechanism we develop to extract kernel execution patterns is composed of three steps: (1) build the kernel execution list over time; (2) identify the kernel signature; and (3) pass the future kernel information to the optimizer.

The kernel pattern extractor samples the performance counters at runtime and stores them in a reduced format. These performance counters are then used by our power and performance predictor. The execution ordering list is dynamically extracted when our framework first encounters the benchmark. At this initial stage, our MPC framework simply runs PPK while it dynamically extracts the pattern.

The pattern extractor implements the dynamic pattern extractor as proposed by Totoni et al. [28]. It identifies different kernels through their signature, extracts the execution pattern once it observes a repetitive behavior, and stores the ordering along with the performance counters.

To find the kernel signatures at run-time, we first reduce the number of performance counters to reduce the runtime compute and storage overheads. This is done by clustering the counters that are more correlated in a similar fashion as Zhu and Reddi [29]. Based on the clustering, we select eight representative performance counters that reflect any input data and kernel characteristics, as presented in Table III. Our pattern extractor stores eight of these performance counters along with the kernel time and power as double-precision values, which accounts for 80 bytes, for each dissimilar kernel.

Next, we approximate kernels with similar performance by binning their counter values according to the following

[2]Ratio of predicted change in energy to change in configuration.

TABLE III: GPU performance counters.

Name	Description
GlobalWorkSize	Global work-time size of the kernel.
MemUnitStalled	Percentage of GPUTime the memory unit is stalled.
CacheHit	Percentage of fetch, write, atomic, and other instructions that hit the data cache.
VFetchInsts	Average number of vector fetch instructions from video memory executed per work-item.
ScratchRegs	Number of scratch registers used.
LDSBankConflict	Percentage of GPUTime LDS is stalled by bank conflicts.
VALUInsts	Average number of vector ALU instructions executed per work-item.
FetchSize	Total kB fetched from video memory.

formula: $\text{bin}_i = \lfloor \log u \rfloor, \forall u \in S$, where S is the eight performance counters. The tuple $(\text{bin}_1, ...\text{bin}_k)$ is the signature.

The kernel signature and the execution ordering together maintain an indexed list of kernels. In successive iterations, the pattern extractor identifies which kernel signature to expect in the future and passes the corresponding performance counters to the prediction model, and the expected instruction count to the optimizer. It also dynamically updates the stored kernel performance counter values based on the performance counter feedback of the last executed kernel.

3) Performance and Power Predictor: The performance and power predictor uses an offline trained model that predicts the power and performance of a kernel. It takes as inputs the performance counters of future kernels from the kernel pattern extractor and the corresponding hardware configuration, and provides the power and performance estimates of a kernel for any desired hardware configuration.

Our performance and power model uses machine learning to model the behavior of the integrated GPU. We use a Random Forest regression algorithm [30] to capture the GPU power and performance behavior. Random Forest is an ensemble learning method that creates multiple regression trees for each random subset of the training data. The predicted class is the mean prediction from these individual regression trees. We selected Random Forest because it gave the highest accuracy among other learning algorithms.

For the kernel performance and power prediction, Random Forest uses the kernel-level GPU performance counters, kernel execution time, and GPU (including NB) power numbers for several benchmark suites executed under different GPU/NB configurations. Since the GPU and NB share the same voltage plane, the GPU power numbers also capture the NB power and the effect of changing NB configurations. The model is trained offline and the system-level software implements the predictor. The accuracy of this model is described in Section VI-D. For CPU power prediction, we use a normalized V^2f model because the CPU usually busy waits while the kernel is executing.

4) Adaptive Horizon Generator: The choice of a horizon length H is a tradeoff between the quality of the solution and the computation overhead of the algorithm. The overhead may be particularly problematic for applications with short GPU kernels separated by short CPU times. Even with our polynomial time MPC algorithm, the value of H must be carefully chosen to avoid significant runtime overheads for these applications.

To address this issue, we propose to dynamically adapt the value of H on a per-kernel basis at runtime. The adaptive horizon generator determines the horizon length H_i for each upcoming i^{th} kernel such that the total performance loss (the MPC overhead plus the performance loss due to MPC approximations and imperfect predictions) remains bounded.

To determine the horizon H_i for each i^{th} kernel, we make use of the information gathered on the first invocation of the application, namely: (1) the number of kernels N, (2) the average per-kernel horizon length $\mathbb{N}$ calculated from the search order, and (3) the total time to run PPK during the initial invocation T_{PPK}.

The adaptive horizon generator determines a horizon length H_i of the present i^{th} kernel based on the estimated MPC overhead ($H_i \times \frac{\mathbb{N}}{N} \times T_{PPK}$), the total execution times of the previous $i-1$ kernels ($\sum_{j=1}^{i-1} T_j$), the total MPC optimization overhead incurred for the previous $i-1$ kernels ($\sum_{j=1}^{i-1} T_{MPC,j}$), and the estimated execution time of the present i^{th} kernel (T_{total}/N). We attempt to bound the performance penalty relative to the baseline Turbo Core execution time so far, including the current kernel ($i \times T_{total}/N$), to a factor α, as shown below.

$$\frac{H_i \times \frac{\mathbb{N}}{N} \times T_{PPK} + \sum_{j=1}^{i-1}(T_j + T_{MPC,j}) + T_{total}/N}{i \times T_{total}/N} \leq 1+\alpha$$

Solving for H_i, we get:

$$H_i \leq \frac{N}{\mathbb{N}} \frac{(1+\alpha-\frac{1}{i})\frac{i \times T_{total}}{N} - \sum_{j=1}^{i-1}(T_j + T_{MPC,j})}{T_{PPK}}$$

We take the floor of H_i to create an integer value, and further bound H_i to be between 0 and N.

V. Experimental Methodology

In this paper, we use an AMD A10-7850K APU as our experimental platform. We use this APU in our studies because, due to its more stringent thermal constraints, it more aggressively manages power compared to discrete GPUs. The core concepts, observations, and insights from this work are also applicable to other heterogeneous processors.

We implemented the MPC framework on the host CPU of the AMD A10-7850K APU running at the hardware configuration of [P5, NB0, DPM0 and 2 CUs]. The CPU runs the MPC algorithm between GPU kernel invocations. While in a real implementation, there may be an idle CPU available to run the algorithm during CPU phases between the GPU kernels, we assume a worst-case scenario in which the GPU kernel invocations occur back-to-back, or a CPU is not available to run the algorithm during the CPU phase. In our studies, the horizon length generator attempts to limit the maximum performance loss to an α of 0.05 (5%).

In order to simulate our approach as well as competing schemes, we captured performance and power data on the

TABLE IV: Benchmarks with their execution pattern.

Category	Benchmarks	Benchmark Suite	Reg. Exp.
Regular	mandelbulbGPU	Phoronix [31]	A^{20}
	NBody	AMD APP SDK [32]	A^{10}
	lbm	Parboil [33]	A^{10}
Irregular w/ repeating pattern	EigenValue	AMD APP SDK [32]	$(AB)^5$
	XSBench	Exascale	$(ABC)^2$
Irregular w/ non-repeating pattern	Spmv [17]	SHOC [34]	$A^{10}B^{10}C^{10}$
	kmeans	Rodinia [18]	AB^{20}
Irregular w/ kernels varying with input	swat	OpenDwarfs [35]	No pattern. Multiple iterations of a same kernel varying with input arguments.
	color	Pannotia [36]	
	pb-bfs	Parboil [33]	
	mis	Pannotia [36]	
	srad	Rodinia [18]	
	lulesh	Exascale	
	lud	Rodinia [18]	
	hybridsort	Rodinia [18]	

AMD hardware for 336 APU hardware configurations by varying the CPU, NB and three out of five GPU DVFS states as shown in Table I, and changing the number of active GPU CUs from 2 to 8 in steps of 2. We use AMD CodeXL to capture the runtime GPU performance counters and measure CPU and GPU power from the APU's power management controller at 1ms intervals. The NB power is included in the GPU measurement, since they share the same voltage rail. This extensive power and performance information, which is captured at run-time for the individual kernels for each of the benchmark suites described in the next subsection, permits accurate comparison of the performance and energy use of different power management schemes with respect to the baseline AMD Turbo Core approach.

A. GPGPU Benchmarks

We study 73 benchmarks from 9 popular benchmark suites and sample 15 of them (Table IV) that have wide-ranging behavior and utilize the hardware in different ways. Within the 73 benchmarks we studied, we found that 75% are irregular and 44% of the kernels varied significantly with input. To represent such a distribution, we categorize our benchmarks according to their kernel execution pattern. Regular benchmarks have a single kernel that iterates multiple times; we include these to show that MPC does not degrade performance or energy efficiency for regular applications. Irregular applications are categorized into the ones with repeating and non-repeating kernel patterns, and those that vary with inputs.

B. Baseline Schemes

We report the energy and performance improvements with respect to the default Turbo Core scheme in the AMD A10-7850K [37]. Turbo Core is a state-of-the-practice technique that balances power and performance under thermal constraints. It controls the DVFS states based on the recent resource utilization, and shifts power between the GPU and CPU based on their recent load. For these GPGPU applications, the CPU busy waits while the GPU is executing the kernel. Therefore, Turbo Core does not drop the CPU DVFS states as long as the system stays within its TDP.

We also compare our MPC method to the PPK and TO schemes described in Section III. PPK represents the state-of-the-art predictive techniques for GPGPU benchmarks that do not consider future kernel behavior [8], [9], [19], while TO is an impractical scheme that demonstrates what is theoretically possible. Furthermore, since the CPU is mostly busy-waiting, due to the nature of the available benchmarks, we also compare the energy savings both with and without the CPU energy to provide a fair assessment.

Upon encountering the benchmark for the first time, all the schemes run PPK, while dynamically extracting the kernel execution pattern. At this stage, our framework starts with no stored knowledge. The very first kernel is run at fail-safe since no performance counters are available to predict its power and performance. Subsequently, PPK uses the previous kernel's performance counters to predict the next kernel's energy-optimal configuration.

VI. Results

In this section, we first show the benefits of MPC after the initial run of the application has been performed, and then explore how the initial energy and performance losses of running PPK the first time are amortized over multiple executions, as encountered in real-world applications. Unless otherwise stated, all of our results include the energy and performance overheads of the MPC and PPK optimizations.

A. Energy-Performance Gains

Figure 8 compares the energy savings and performance impact of PPK and MPC over AMD Turbo Core. MPC fares similarly to PPK for the three regular benchmarks with a single repeating kernel. However, the differences are pronounced for the irregular benchmarks whose complex patterns benefit from additional future knowledge. Here, MPC considers the future kernel behavior and mitigates the performance losses of looking only a single kernel into the future, while simultaneously saving energy. Overall, including the MPC overheads, MPC achieves a 24.8% energy savings over Turbo Core with a 1.8% performance loss. Except *srad*, MPC achieves a maximum performance loss of 3.8% for *hybridsort*. This is because MPC adaptively tunes the MPC horizon and restricts the total performance loss to 5%. The 15.7% performance loss for *srad* represents a worst-case scenario for our MPC approach with imperfect prediction. Here, the prediction model mispredicts during the last phases of *srad*, and MPC is unable to recover from the performance loss.

Figure 9 shows the results of MPC with respect to PPK, which include the optimization overheads. Unlike the PPK approach described in Section II-E with perfect performance and power prediction, for a fair comparison, this version uses Random Forest for power and performance prediction as with MPC. Among the regular benchmarks, PPK works well for *mandelbulbGPU* and *NBody* because the same kernel

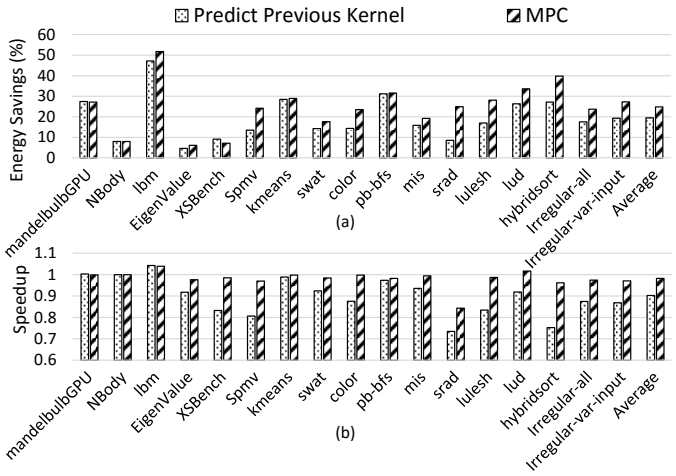


Fig. 8: PPK and MPC (a) energy savings and (b) speedup over AMD Turbo Core.

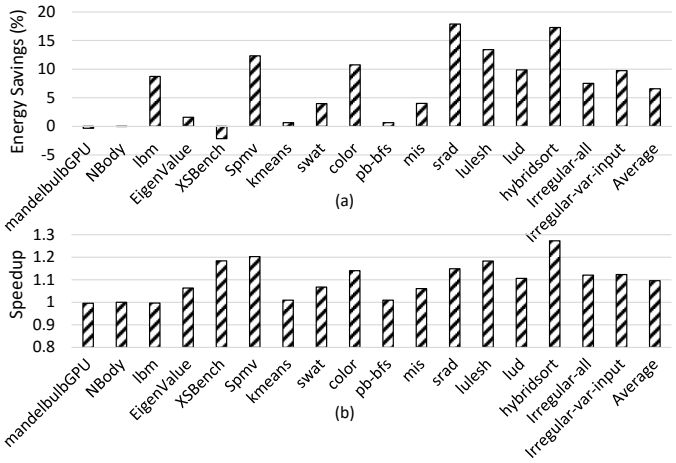


Fig. 9: MPC (a) energy savings and (b) speedup over PPK.

is iterated and the kernels are input independent. For these benchmarks, MPC does not show an advantage.

MPC significantly outperforms PPK for the 12 irregular benchmarks, where PPK often mispredicts the next kernel behavior, achieving 12% greater performance than PPK while simultaneously reducing energy by 7.5%. For these benchmarks, PPK suffers an 8–26% performance loss compared to AMD Turbo Core (Figure 8). This is due to next kernel misprediction in conjunction with the inability to proactively change decisions based on future kernel behavior. In contrast, MPC foresees the ability to catch up on the lost performance due to mispredictions in future kernels. For example, for *srad*, MPC outperforms PPK by 15%. MPC performs particularly well for the irregular benchmarks with kernels with varying input, outperforming PPK by 12.3% while reducing energy by 9.7%. For *XSBench*, MPC consumes more energy than PPK by choosing higher power configurations to reduce the performance loss. Overall, MPC outperforms PPK by 9.6% while reducing energy by 6.6%.

The CPU's contribution to the overall MPC energy savings over Turbo Core is 75%, while the GPU contributes 25%.

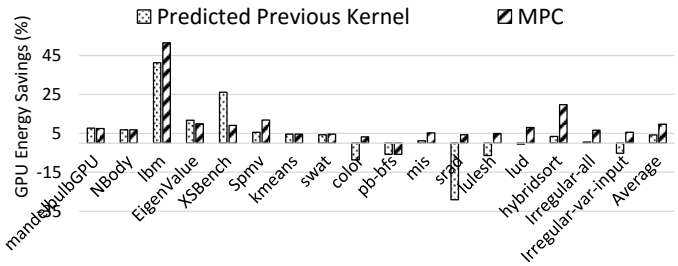


Fig. 10: GPU energy savings over AMD Turbo Core.

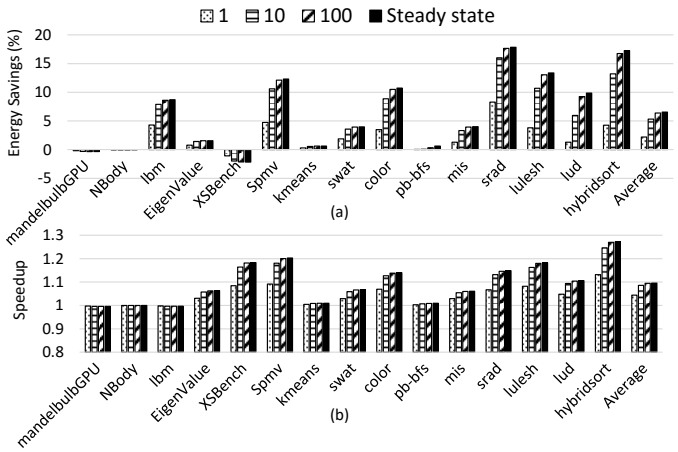


Fig. 11: MPC (a) energy savings, and (b) speedup relative to PPK when the benchmarks are re-executed the specified number of times after the initial execution.

This is because MPC intelligently lowers the CPU state as it does not improve the kernel execution time, whereas Turbo Core keeps the CPU at a higher DVFS state as long as the system is operating within its TDP limit. For this reason, we also show the GPU energy savings of MPC over Turbo Core in Figure 10. These energy savings also includes the static energy overhead of the GPU during MPC optimization.

The highest savings (51%) is achieved for *lbm* because its kernels exhibit peak behavior. For other benchmarks, the savings is not as large, but still significant (3-20%), which leads to an overall energy savings of 10%. For *EigenValue* and *XSBench*, PPK shows higher GPU energy savings than its chip-wide savings. This is because PPK lowers the CPU and GPU power states while significantly increasing the execution time, thereby resulting in higher CPU energy. Compared with PPK, MPC achieves an average GPU energy savings of 5.1% while simultaneously improving performance by 9.6%.

B. Amortization of Initial Losses

Our approach benefits from repeated application execution to achieve gains. The initial losses of running PPK for the first execution can be amortized over these repeated executions. Figure 11 shows the energy savings and performance loss of MPC compared to PPK when the benchmarks are re-executed the specified number of times after the initial execution. The energy savings and performance loss includes the associated overheads. The steady state value is the ideal case with no initial losses during the profiling. Non-negligible gains are

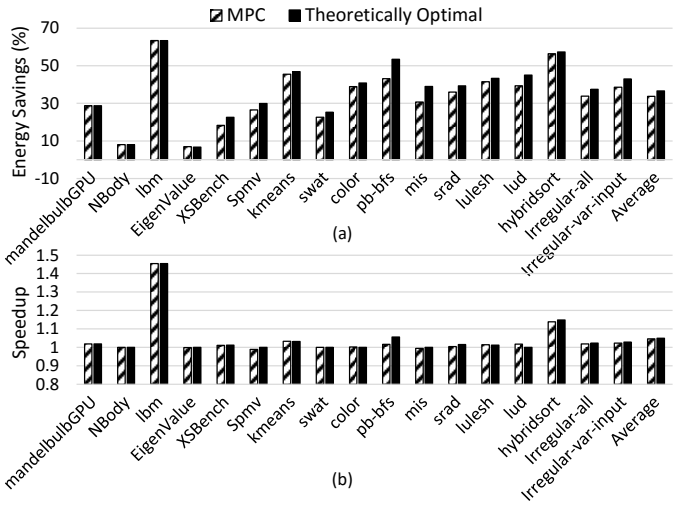


Fig. 12: Comparison with Theoretical Limit. (a) Energy savings and (b) speedup over AMD Turbo Core.

observed with just a single repeated execution, and most of the full gains are observed after only ten re-executions, indicating that MPC can significantly improve the energy efficiency of real-world workloads that repeatedly re-execute.

C. Comparison with Theoretical Limit

In this section, we explore how closely our polynomial-time heuristic MPC approach matches the theoretically achievable savings by comparing against the exponential-time Theoretically Optimal (TO) scheme. We assume perfect prediction, no MPC overhead, exhaustive search of all hardware configuration for each kernel, and a horizon length of all kernels. Figure 12 shows the results.

As expected, MPC performs similarly to TO for regular benchmarks. In general, MPC benefits from looking into the future behavior of all the kernels, and thus achieves near-optimal energy savings and performance gains. In particular, *pb-bfs*, *mis* and *lud* show lower energy savings than TO, while *EigenValue*, *mis* and *Spmv* suffer a slight performance loss. This is because the effectiveness of MPC is highly sensitive to its search order, which is derived based on the sub-optimal PPK-based profiling. Overall, MPC achieves 92% of the maximum theoretical energy savings and 93% of the potential performance gain.

D. Ramification of Prediction Inaccuracy

The Mean Absolute Percentage Errors of our Random Forest prediction model over the 15 benchmarks are 25% and 12% for performance and power respectively. The high performance error is due to diverse performance scaling trends and the presence of outliers with unexpected performance behavior. In this section, we examine the potential loss in energy savings by our RF-based MPC compared to a MPC using a perfect prediction model. We consider a horizon length equal to the number of kernels and exclude the MPC overhead.

Figure 13 compares our Random Forest based MPC implementation (RF) with MPC implementations based on the accuracy of recently published prediction models. Err_15%_10% assumes prediction inaccuracies of 15% and 10% for performance and power respectively, as reported by Wu et al. [38]. Similarly, Err_5% considers prediction inaccuracy of 5%, as reported for Paul et al. [8]. A prediction model with no errors is represented by Err_0%. To implement these prediction models, we assume a half random normal distribution [39], with its absolute mean equal to the corresponding average error.

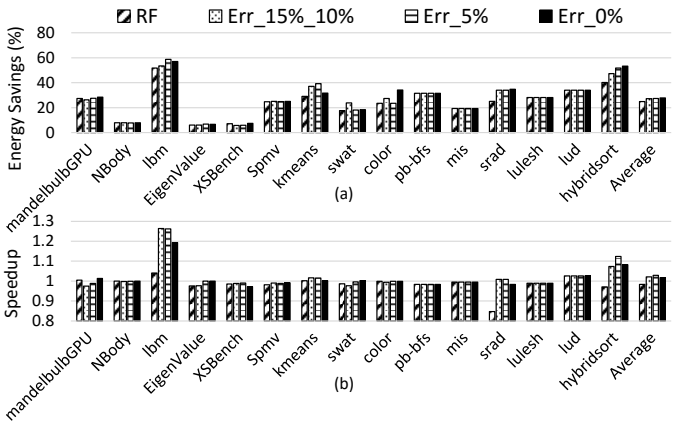


Fig. 13: Ramification of prediction inaccuracy on energy-performance tradeoff.

From Figure 13, RF behaves similar to Err_15%_10%. RF is better for *mandelbulbGPU* and *XSBench*, while Err_15%_10% is better for *kmeans*, *swat* and *srad*. On average, the energy savings of other models range from 27-28%, while RF's savings is 25%. Similarly, other prediction models improve performance by 1.7-3%, while RF decreases performance by 1.7%. The reason that the energy and performance results are not highly sensitive to prediction accuracy is that MPC relies on the prediction models far less (a factor of 65×) than exhaustive search. It also takes the runtime performance as feedback and thus further rectifies the impact of these mispredictions by dynamically updating the performance headroom. The result is comparable energy savings with minor differences in performance.

E. MPC Overheads and Horizon Length

Figure 14 shows the MPC energy and performance overheads with respect to Turbo Core when adapting the horizon length for an α of 0.05 (5%). The average energy overhead is 0.15% (maximum of 0.53% for *Spmv*) with a performance overhead of 0.3% (maximum of 1.2% for *Spmv*). The overheads consider a worst case situation when kernels appear back-to-back with no CPU phases in between, or when there are no available CPUs to run the algorithm during CPU phases. In practice, GPGPU application kernels may be separated by CPU phases with an available CPU, which can hide the MPC overheads. As a result, the actual overheads will be lower, permitting longer horizon lengths to improve performance.

Figure 15 shows the average MPC horizon length as a percentage of N, the total number of kernels in an application. Benchmarks *NBody*, *lbm*, *EigenValue* and *XSBench*

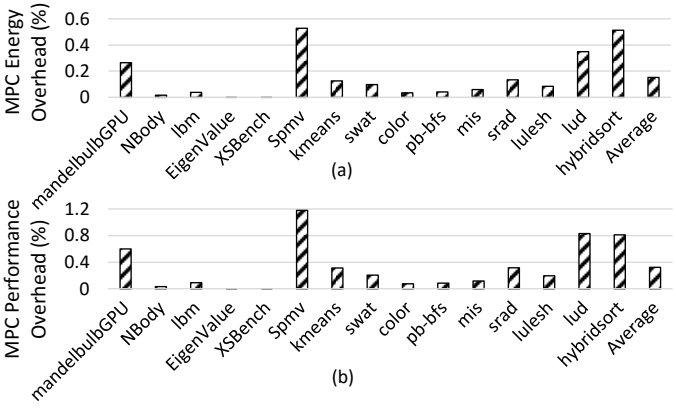


Fig. 14: MPC (a) energy and (b) performance overheads with respect to Turbo Core.

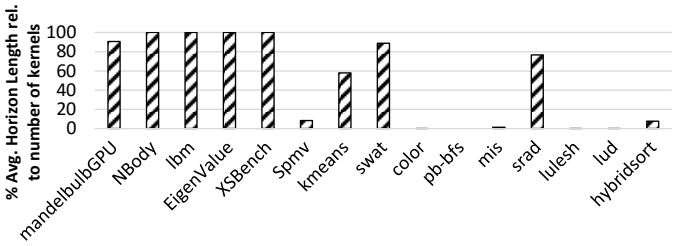


Fig. 15: Average MPC horizon as a percentage of the total number of kernels.

have long kernels, which permits MPC to explore the full horizon. For *MandelbulbGPU*, *kmeans* and *swat*, the horizon length generator initially selects a low horizon length before determining that there is enough performance margin to use the full horizon. The full horizon is initially selected for *srad*, but lowered when encountering a performance loss due to misprediction. For the remaining benchmarks, the horizon length generator shrinks the horizon length significantly to limit the overheads since they have shorter kernel lengths.

We compare our adaptive horizon MPC scheme with one that uses full horizon. When ignoring overheads, the full-horizon MPC approach reduces energy by only 2.6% compared to our adaptive scheme, with similar performance impact. When the MPC overheads are included, the full-horizon scheme achieves a 15.4% energy savings over Turbo Core, with a performance loss of 12.8%, compared to 24.8% energy savings with a 1.8% performance loss for the adaptive scheme.

VII. Related Work

Among the *reactive* power-performance optimization studies, Pegasus captures the historical latency statistics to dynamically control the CPU DVFS states [40], while Sethia and Mahlke's Equalizer [9] monitors the performance counters and reactively tunes the GPU knobs to operate under *performance boost* or *energy efficiency* mode.

Other prior work [41], [42], [43] propose analytical estimation models, while [38], [44], [45], [46] present learning or statistical models. Optimizing power efficiency using these estimation models has been proposed by [19], [46], [47], [48], [49]. All of these *locally predictive* studies use the past behavior to represent the immediate phase, while Chen et al. [12] predicts the execution time of the immediate next phase of the embedded applications. These schemes do not use feedback to fine-tune, and therefore are unable to recover from the past performance losses.

Within the feedback-driven power management schemes, Paul et al. [8] train linear regression models to predict performance and power sensitivities and use two levels of tuning to adapt based on the past performance trend at the kernel level, without across-kernel considerations. Our PPK scheme represents such state-of-the-art future agnostic schemes. For applications with irregular throughput phases, we demonstrate significant reduction in performance loss with substantial energy savings over such schemes.

VIII. Conclusion

This paper presents a dynamic power management scheme for GPGPU applications using Model Predictive Control (MPC). MPC anticipates future kernel behavior and makes proactive decisions to maximize energy efficiency with minimum impact on performance. We devise a variant of MPC that uses greedy and heuristic approximations and adaptively tunes the horizon length to permit a low overhead practical runtime implementation. Our scheme achieves significant energy savings with negligible performance loss compared to the AMD Turbo Core power manager, and both energy savings and performance improvement over current history-based approaches.

References

[1] S. Nussbaum, "AMD "Trinity" APU." Presented at Hot Chips, August 2012.

[2] P. Dongara, L. Bircher, and J. Darilek, "AMD Richland Client APU." Presented at Hot Chips, August 2013.

[3] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge," *IEEE Micro*, vol. 32, pp. 20–27, March 2012.

[4] E. Rotem, R. Ginosar, C. Weiser, and A. Mendelson, "Energy Aware Race to Halt: A Down to EARtH Approach for Platform Energy Management," *Computer Architecture Letters*, vol. 13, pp. 25–28, Jan–June 2012.

[5] E. Rotem, "Intel Architecture, Code Name Skylake Deep Dive: A New Architecture to Manage Power Performance and Energy Efficiency." Presented at Intel Developer Forum, August 2015.

[6] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim, "Workload and Power Budget Partitioning for Single-chip Heterogeneous Processors," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[7] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2013.

[8] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, "Harmonia: Balancing Compute and Memory Power in High Performance GPU," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2015.

[9] A. Sethia and S. Mahlke, "Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2014.

[10] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "Multi-Kernel Auto-Tuning on GPUs: Performance and Energy-Aware Optimization," in *Proc. of the Int'l. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, 2015.

[11] A. McLaughlin, I. Paul, J. L. Greathouse, S. Manne, and S. Yalamanchili, "A Power Characterization and Management of GPU Graph Traversal," in *Workshop on Architectures and Systems for Big Data (ASBD)*, 2014.

[12] T. Chen, A. Rucker, and G. E. Suh, "Execution Time Prediction for Energy-efficient Hardware Accelerators," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2015.

[13] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-Mark, A Benchmark Suite for CPU-GPU Collaborative Computing," in *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2016.

[14] O. Kayiran, A. Jog, M. Kandemir, and C. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[15] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2014.

[16] J. Lee, P. P. Ajgaonkar, and N. S. Kim, "Analyzing Throughput of GPGPUs Exploiting Within-Die Core-to-Core Frequency Variation," in *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[17] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format," in *Proc. of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2009.

[19] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili, "Coordinated Energy Management in Heterogeneous Processors," in *Proc. of the Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[20] M. R. Garey and D. S. Johnson, "" Strong " NP-Completeness Results: Motivation, Examples, and Implications," *Journal of ACM*, vol. 25, pp. 499–508, July 1978.

[21] A. Majumdar, J. L. Setter, J. R. Dobbs, B. M. Hencey, and D. H. Albonesi, "Energy-Comfort Optimization using Discomfort History and Probabilistic Occupancy Prediction," in *Proc. of the Int'l. Green Computing Conference (IGCC)*, 2014.

[22] R. Van Der Linden and A. P. Leemhuis, "The Use of Model Predictive Control for Asset Production Optimization: Application to a Thin-Rim Oil Field Case," in *SPE Annual Technical Conference and Exhibition*, Society of Petroleum Engineers, 2010.

[23] T. Le, H. L. Vu, Y. Nazarathy, B. Vo, and S. Hoogendoorn, "Linear-Quadratic Model Predictive Control for Urban Traffic Networks," in *Proc. of the Int'l Symp. on Transportation and Traffic Theory*, 2013.

[24] A. Marquez, C. Gomez, P. Deossa, and J. Espinosa, "Infinite Horizon MPC and Model Reduction Applied to Large Scale Chemical Plant," in *Proc. of the Robotics Symposium, Latin American and Colombian Conference on Automatic Control and Industry Applications (LARC)*, 2011.

[25] J. Löfberg, *Minimax Approaches to Robust Model Predictive Control*, vol. 812. Linköping University Electronic Press, 2003.

[26] M. H. Chauhdry and P. B. Luh, "Nested Partitions for Global Optimization in Nonlinear Model Predictive Control," *Control Engineering Practice*, vol. 20, no. 9, pp. 869 – 881, 2012.

[27] Y. Wang and S. Boyd, "Fast Model Predictive Control Using Online Optimization," *IEEE Trans. on Control Systems Technology*, vol. 18, pp. 267–278, March 2010.

[28] E. Totoni, J. Torrellas, and L. V. Kale, "Using an Adaptive HPC Runtime System to Reconfigure the Cache Hierarchy," in *Proc. of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[29] Y. Zhu and V. J. Reddi, "High-performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems," in *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2013.

[30] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[31] "PHORONIX TEST SUITE." http://www.phoronix-test-suite.com/.

[32] "APP SDK - A Complete Development Platform." http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/.

[33] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, March 2012.

[34] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *Proc. of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[35] W. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 Dwarfs: A Work in Progress," in *Proc. of the Int'l Conf. on Performance Engineering (ICPE)*, 2012.

[36] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2013.

[37] Advanced Micro Devices, Inc, *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors*, February 2015.

[38] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU Performance and Power Estimation Using Machine Learning," in *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[39] R. H. Byers, *Half-Normal Distribution*. John Wiley & Sons, Ltd, 2005.

[40] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-scale Latency-critical Workloads," in *Proc. of the 41st Annual Int'l Symp. on Computer Architecture*, ISCA '14, pp. 301–312, 2014.

[41] B. Su, J. L. Greathouse, J. Gu, M. Boyer, L. Shen, and Z. Wang, "Implementing a Leading Loads Performance Predictor on Commodity Processors," in *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)*, 2014.

[42] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2010.

[43] C. Isci and M. Martonosi, "Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2003.

[44] B. C. Lee and D. M. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," in *Proc. of the Int'l Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[45] C. Zhang, A. Ravindran, K. Datta, A. Mukherjee, and B. Joshi, "A Machine Learning Approach to Modeling Power and Performance of Chip Multiprocessors," in *Proc. of the Int'l Conf. on Computer Design (ICCD)*, 2011.

[46] G. Dhiman and T. S. Rosing, "Dynamic Power Management Using Machine Learning," in *Proc. of the Int'l Conf. on Computer-Aided Design (ICCAD)*, 2006.

[47] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[48] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online Power-performance Adaptation of Multithreaded Programs Using Hardware Event-based Prediction," in *Proc. of the Int'l Conf. on Supercomputing (ICS)*, 2006.

[49] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, "PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2014.