Energy-Aware Workload Consolidation on GPU

Dong Li[§*] Surendra Byna[+*] Srimat Chakradhar[*]

{lid, chak}@nec-labs.com, sbyna@lbl.gov
*NEC Laboratories America, Princeton, NJ
§ Oak Ridge National Lab, Oak Ridge, TN
+ Lawrence Berkeley National Lab, Berkeley, CA

Abstract—**Enterprise workloads like search, data mining and analytics, etc. typically involve a large number of users who are simultaneously using applications that are hosted on clusters of commodity computers. Use of GPUs for enterprise computing is challenging because of poor performance and higher energy consumption compared to running enterprise workloads on CPUs. In this paper, we show that the GPU work consolidation can improve system throughput and results in significant energy savings over multicore CPUs. We develop a novel runtime framework that dynamically consolidates instances from different workloads from multiple user processes into a single GPU workload. However, arbitrary consolidation of GPU workloads does not always lead to better energy efficiency. We use new GPU performance and power models to make predictions for potential workload consolidation alternatives and identify useful consolidations. Our experiments on a variety of workloads (that perform poorly on a GPU compared to well optimized multicore CPU implementations) show that the proposed framework for GPU can provide 2X to 22X energy benefit over a multicore CPU.**

Keywords-Power aware computing, GPU computing, Workload consolidation

I. INTRODUCTION

Many data parallel applications (especially scientific computing applications) use general purpose graphical processing units (GPGPUs or commonly known as GPUs) to achieve impressive speedups, in the range of 100X for many data parallel applications [25]. The three of the top five supercomputers in the world today already mix CPUs with GPUs to boost performance, and to make the supercomputers energy efficient. However, use of GPUs for enterprise computing is challenging because (a) each user request does not usually have enough work to offset the overheads of off-loading to a GPU, and (b) the GPU compute fabric and memory architecture is also not a good fit for enterprise kernels like search, sorting, encryption etc. Enterprise workloads typically involve a large number of users who are simultaneously making requests to applications that are hosted on clusters of commodity computers. Since the user requests typically have small amount of work, the execution time of these workloads on a GPU is more than the execution time on a multicore CPU. Moreover, the power consumption of a typical GPU is also much higher than that of a multicore CPU. For example, NVIDIA Tesla GPUs consume more than 250 Watts at peak, more than twice the peak power consumption of a multicore CPU [2]. Even though power-related innovations by GPU vendors at the hardware level may happen in the future, the relative big power consumption of GPU will still prevent it from the adoption in some cases, e.g., high-performance embedded environments [29].

When the GPU is unable to provide impressive acceleration for many popular enterprise workloads, the question of whether the GPU is more energy-efficient than a multicore CPU arises. In this paper, we answer this question by developing a new GPU workload consolidation framework that combines small GPU workloads from multiple users (i.e., form multiple processes) into a single, large workload. We show that consolidation strategy achieves significant energy savings for a variety of enterprise workloads executing on a GPU, as compared to a multicore CPU and to a GPU without consolidation. We assume an environment where there are many users who are simultaneously sending their requests to a set of known applications hosted on enterprise computing platforms. We also assume that user requests collectively include sufficient workload for GPU processing so that there is an opportunity to combine workloads. These assumptions are valid in typical enterprise computing scenarios.

The proposed workload consolidation strategy is different from multi-kernel execution and from CUDA 4.0 GPU sharing features on NVIDIA's Fermi GPUs [16]. The Fermi GPUs can execute multiple kernels but these kernels must be issued from the same process context. In other words, multiple threads of the same process can share a GPU. However, in data center environment, different processes spawn GPU workloads that typically belong to multiple users. Our proposed strategy can consolidate workload instances from different contexts (i.e., processes from different users).

The consolidation strategy is not limited to enterprise computing. Any workloads, including from scientific computing, can potentially benefit from our framework, when GPUs have idle cores or if workloads are unevenly distributed on GPU cores. For example, some workloads (e.g., matrix computation) have scalability limitation [1], where only a fraction of available streaming multiprocessors (SMs) are required to achieve the best performance. These SMs may be released by applications and stay idle wasting energy. Some workloads unevenly utilize GPU resources (e.g., distributing 45 thread blocks on 30 SMs of a GPU), where the lightly-loaded SMs finish earlier and have to wait. With judicious consolidation of these workloads, we can improve system throughput while saving energy.

We develop a framework that dynamically consolidates GPU workload instances from multiple users. Our framework does not rely on CUDA driver and runtime. The

GPU workloads can be the instances of the same application (homogenous) or different applications (heterogeneous). Here, an instance means a workload in the ready state to run. The number of workload kernels to be consolidated is not restricted, if their total resource requirement is not higher than GPU shared resources. The framework uses energy awareness as the main criterion for making decisions on consolidating workload instances. Consolidation has a possibility of lowering throughput due to the contention for shared GPU resources, such as GPU global memory bandwidth, shared memory, register file, and constant memory. Additionally, arbitrary consolidation of user requests may have adverse effect on performance and may lead to poor energy efficiency. Consolidation of workloads leads to increased power due to increased requirement of GPU resources. Since energy consumption is the product of power and execution time, throughput improvement must be higher enough to offset the increased power for achieving energy efficiency. We propose new GPU power and performance prediction models to identify an energy-efficient consolidation of workloads.

In this paper, we make the following contributions:

• We demonstrate that energy consumption of GPUs can be significantly lower than that of CPUs, even for those workloads that perform better on CPU than on GPU if they were run individually;

• We propose a lightweight GPU power model that captures power-critical events for consolidated homogeneous and heterogeneous workloads;

• We develop a GPU performance prediction model that considers the performance impact of thread blocks scheduling;

• We introduce a framework that facilitates workload consolidation on NVIDIA GPUs.

In the rest of the paper, we first provide related work and motivational examples before describing our dynamic framework and GPU performance and power models. We evaluate energy efficiency results of our framework and then conclude.

II. RELATED WORK

Workload Consolidation: The consolidation runtime explored by Guevara et al. [10] is the closely related work to our study. They propose an issue queue, called *cusub* that could be included in CUDA driver, to merge workloads and a prototype implementation shows throughput benefits in merging workloads. Our work differs from the issue queue approach for merging kernels in two major ways: (a) our process-level consolidation framework can consolidate multiple instances (i.e., homogeneous or heterogeneous instances of workloads) without imposing any constraints on CUDA driver and on CUDA runtime; and (b) we use new GPU power and performance models to consolidate workloads for energy savings over a multicore CPU rather than focusing on performance alone. These models are inevitable for the determination of performance and energy benefits. Our consolidated kernels can further take advantage of optimizations like an issue queue in the CUDA driver.

Energy Efficiency on GPU: A few recent studies explored proving energy efficiency of running highly data parallel workloads on GPUs. Ren et al. [12] and Huang et al. [13] prove that energy efficiency of GPUs is high compared to CPUs for matrix multiplication kernels. The matrix multiplication kernels are highly data parallel and it is proven that these kernels achieve high speedups on GPUs compared to CPUs. Energy efficiency with such high speedups is obvious. Takizawa et al. [18] provide a simple GPU energy model for selecting either CPU or GPU to run a batch of workloads on a heterogeneous cluster, with the assumption that GPU power consumption is constant across tasks. However, this assumption is not true for many workloads.

GPU Power Model: Nagasaka et al. [5] set up a power model based on the absolute access number of system components. Ma et al. [19] propose to record 5 specific runtime GPU power signals and apply statistical analysis to estimate GPU power. Hong and Kim [1] propose a power model based on the access rates of a large amount of components and the estimation of max power consumption of each system component.

GPU Performance Model: There have been many recent efforts in predicting GPU performance. Ryoo et al. [21] used Pareto-optimal curve to prune the optimization space of application on GPU based on two metrics (efficiency and utilization). Hong et.al [8] proposed an analytical model to capture the cost of memory operations. Baghsorhki et al. [22] identify how the kernel exercises major microarchitecture features and uses workflow graphs to detect performance bottleneck, based on which they estimate the performance. Kerr et al. [23] leverage Ocelot dynamic compiler infrastructure to instrument applications to collect 37 metrics. Then they use statistical analysis to derive the relationship between program behavior and performance. Liu et al. [24] classify applications into several categories. Based on the specific characteristics of each category, they establish a relationship between problem sizes and performance, taking into consideration the architecture and GPU programming primitives.

All existing GPU power and performance models mentioned above are developed for the case where all SMs are executing identical workloads. With workload consolidation, different SMs may run different workloads and the amount of work by each thread block on the same SM may also be different. Hence, direct application of the existing models is not sufficient for our consolidation strategy. Our models, which work for variant workload combinations (homogeneous and heterogeneous), are a significant improvement to the previous work.

III. MOTIVATION

In this section, we discuss that workload consolidation is performance and energy efficient even for workloads that achieve poor performance on GPUs when compared to multicore CPU. We also show that workload consolidation must be performed judiciously based on the characteristics of workloads.

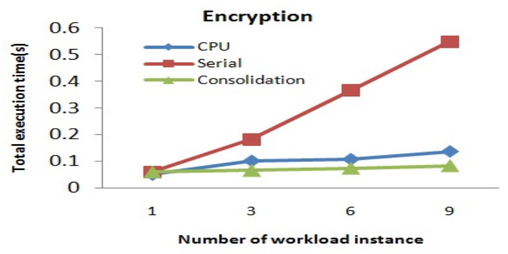
Throughout this paper, we use a heterogeneous compute node with dual socket Intel Xeon E5520 quad-core processors (8 cores in total) and a NIVDIA Tesla C1060 GPU, which has 30 SMs, 4GB global memory. The GPU code is executed with CUDA driver and runtime version 3.0. Except where specifically indicated, the energy reported in the paper refers to the whole system energy consumption. The performance of GPU includes the GPU computation time and data transfer time between host memory and GPU device memory. The performance of CPU refers to the CPU time doing the same computation as GPU. The execution time is the time duration of concurrently running multiple instances, from the point where all instances get started to the point where all instances are finished. Given a fixed number of instances to execute, the execution time also reflects the system throughput. The smaller execution time is, the larger is the system throughput. All the CPU implementations are parallelized and optimized for their best memory access performance.

Table 1 summarizes execution times of various enterprise workloads we used in this paper. Performance of these workloads is either comparable or worse on the GPUs than on the CPUs. Note that these workloads can achieve speedups on the GPU compared to optimized and parallelized CPU implementations, if data sizes are much larger, but the data sizes shown in Table 1 are representative of enterprise computing scenarios.

Table 1. Poor GPU speedup over multicore CPU

Name	Input data size	# of blocks /instance	GPU speedup over CPU	#threads/ block
Encryption [26]	12K	3	0.84	256
Encryption	6K	3	0.15	128
Sorting [27]	6K	6	1.45	256
Search [7]	10K	10	0.48	256
BlackScholes [28]	4096K	1	1.68	256
MonteCarlo [28]	Steps=500K	1	7.0	128

We now show performance and energy efficiency using workload consolidation. Figure 1 compares the total execution time and energy in running multiple instances of encryption. A single encryption instance (using AES algorithm) with input file of 12KB on the GPU has worse performance (16% lower) than on the CPU. The energy consumption of using the GPU in running one instance is 1.5 times more than that of using the CPU. When multiple instances are run on the CPU, they are scheduled by the OS to run on different cores. The execution time and total energy consumption increase for more number of instances. When we run these multiple instances on GPU in conventional fashion (i.e., serially one after another), the



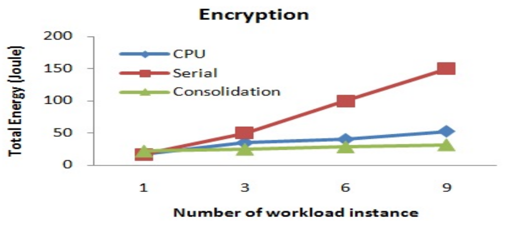


Figure 1: **Benefit with consolidating workloads**

execution time and energy consumption increase almost linearly, which is very energy inefficient. If we consolidate multiple instances into one large GPU kernel, then we observe a significant reduction in total execution time and total energy consumption. In the best case (9 encryption instances), workload consolidation finishes in 68% less execution time with 29% energy savings, compared to the CPU.

Further analysis reveals that a single encryption instance uses only 3 thread blocks and the GPU scheduler distributes thread blocks between SMs in a round-robin fashion. Therefore, an encryption instance uses only three SMs. When we consolidate more instances, GPU power consumption is not linearly increased, as one would expect. Rather, power consumption increases much slowly. This behavior is consistent with a recent study [1]. Also, execution time stays relatively steady as we increase the number of SMs used, because additional encryption instances utilize SMs that were previously idle. Moreover, the additional load on SMs does not introduce resource contention on the shared resources. Consequently, the energy consumption, which is the product of power and execution time, decreases significantly.

Workload consolidation on the CPU does not translate to better performance because of contention for shared resources such as L2 and L3 cache memories. In addition, when a processor core is overloaded with multiple workloads, the CPU suffers from large context switch overhead due to operating system's time slicing between workloads. The GPU has less of these performance concerns, because of its very small caches and unique design of scheduling a large amount of threads to cover any access latency. In addition, our framework merges the workloads from multiple contexts into a single context and thus avoids any context switch overhead.

While consolidation on GPUs offers benefits, consolidating arbitrary workloads will not automatically result in improvement in energy consumption or throughput. For example, consider the following two scenarios.

Scenario 1: Table 2 shows execution time and energy consumption of a single Monte Carlo (MC) instance (45 thread blocks and 50 computation iterations), a single encryption instance (15 thread blocks and 1.0E+5 computation iterations), and a consolidated workload with one instance of each of MC and encryption on GPU.

Scenario 2: Table 3 shows execution time and energy consumption of a BlackScholes instance (45 thread blocks and 1000 computation iterations), a search instance (15 thread blocks, 6E+6 computation iterations), and a consolidated workload consisting of one instance each of search and BlackScholes.

Table 2. Workload consolidation results with scenario 1

Workload	Time (s)	Energy (KJoule)
Single MC	62.4	25.6
Single encryption	19.5	7.03
MC+encryption	84.6	33.5

Table 3. Workload consolidation results with scenario 2

Workload	Time (s)	Energy (KJoule)
Single BlackScholes	26.4	12.2
Single search	49.2	19.2
BlackScholes+Search	58.7	26.7

From these two scenarios, we can observe conflicting execution time and energy consumption trends for the consolidated workloads. In Scenario 1, the execution time of consolidated kernel (84.6s) is larger than the sum of the execution times of both (MC and encryption) instances. The energy consumption is larger than the sum of the individual consumptions. We do not see any energy benefit of this workload consolidation and lost some throughput. In Scenario 2, the execution time of consolidated workload (58.7s) is smaller than the sum of the execution time of two workloads and a little longer than that of the larger workload (i.e., Search), which results in energy savings. The energy consumption of consolidated kernel is also less than the sum of the individual consumptions. Hence, consolidation of random instances does not guarantee energy savings or higher throughput. Hence, it is necessary to judiciously consolidate kernels taking into account important factors like execution time and power consumption of the different instances.

IV. ENERGY-AWARE WORKLOAD CONSOLIDATION FRAMEWORK

This section describes our energy-aware consolidation framework. The main functions of the framework, shown in Figure 2, are to receive GPU workloads from multiple users, to consolidate them into a large kernel, and then to execute the consolidated kernel on the GPUs. The framework uses performance and power prediction models (described in Sections V and VI) to estimate the execution time and the power consumption of consolidation. If the framework determines that consolidation is not beneficial, then the instances are executed either on CPU or on GPU individually based on their individual performance and energy consumption. We assume that CPU performance and energy profiles are available for such comparison. If estimated beneficial, a precompiled consolidated template is selected to run combinations of workloads on GPU.

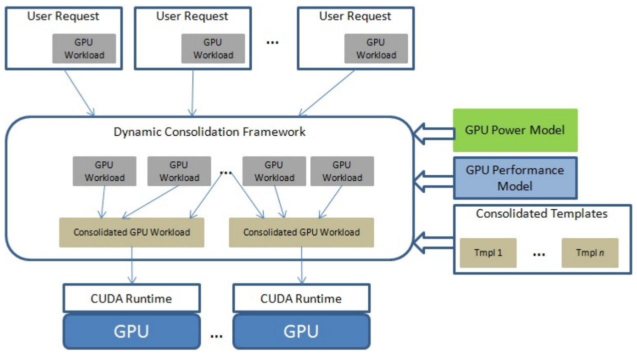


Figure 2: An overview of Energy-aware consolidation framework

A precompiled template is a CUDA kernel that implements a set of consolidated workloads. The templates are manually pre-designed after detailed performance analysis of workload kernels, their frequency of usage and typical data size from data center traces. The templates are parameterized to run multiple instances of consolidated workloads. A template is independent of block partitioning between workloads. They are able to handle the workloads, which are reused often but with different grid configurations. Typically a template is implemented by renaming variables to prevent name collisions, updating the indexes for data accesses, and adding if-else control flow to distribute blocks between SMs. The generation of templates can be automated with a source-to-source compiler.

We consolidate workloads at the granularity of thread blocks. In particular, two or more instances of multiple workloads are executed by the sum of the number of blocks in each kernel request. Multiple workloads can be executed in parallel if their blocks are mapped to different SMs. Multiple workloads can also be interleaved at the block level, if their corresponding blocks are scheduled on the same SM. We do not consolidate workload at the thread level to avoid violating the SIMD lockstep execution of threads in the same warp by inserting control flow divergence. Note that, workload consolidation is not feasible under certain situations, especially, if consolidated workload exceeds the limitation of shared resources, such as the number of registers and shared memory provided to each SM.

The frontend is a shared library, loaded into applications to intercept specific CUDA Runtime API calls. The frontend also sets up a communication channel with the backend, through which the frontend informs the backend of API type and arguments. The backend is a daemon, launched before any workload execution. The backend listens for connection requests from frontend instances and then communicates with the frontends to receive CUDA API information. It is the backend that really conducts the CUDA API calls and kernel calls.

When a memory operation of CUDA API is called from the workload, the front-end related to that kernel intercepts it and passes the API arguments to the backend. The backend carries out the API operations and return the results (e.g., a pointer pointing to the allocated memory region). For the memory copy operations (either from host memory to device global memory, or vice versa), the backend cannot directly copy data between the frontend process context and the device global memory, because the backend and the frontend belong to different process contexts. So the backend copies the data into a local pre-allocated buffer, and then copies the data from the buffer to the destination.

The frontend instances have to be independent of workload implementation so that the frontend can be easily used by any workload without changes. To do this, the frontend intercepts CUDA driver API calls, including (1) *cudaConfigureCall*, where the backend can obtain the execution configuration, such as block and grid dimensions; (2) *cudaSetupArgument*, where the backend can obtain argument information for the workload kernel calls; (3) *cudaLaunch*, where multiple frontends inform the backend of launching the template. It also intercepts GPU memory related operations, such as memory memory allocation (*cudaMalloc*) and memory copy (*cudaMemcpy*). Note that since all memory operations are conducted by the backend, the kernel call arguments are completely valid in the context of the backend.

Our run-time consolidation does have overheads. The main overhead comes from the memory copy operations between the frontends and the backend pre-allocated buffer, the communication costs between the frontends and the backend, and synchronization costs between multiple frontends. To mitigate the impact of overheads, we introduce several optimizations to reduce or offset the overhead.

To reduce communication costs and synchronization costs, we introduce coordination among frontends in consolidating homogeneous workloads. In particular, the framework randomly selects a leader frontend for homogeneous workloads. Then only the leader frontend communicates with the backend. This strategy reduces severe communication overhead.

We also implemented enforcing application specific optimizations in the backend when possible. For example, AES encryption algorithm has large amount of constant data [11] that can be reused by any of its kernels. We provide an API to load reusable data to the GPU memory only once and let multiple kernels use that data.

There are other possible optimizations to reduce communication costs. For example, to transfer the kernel call arguments from the frontend to the backend, instead of transferring them one by one whenever *cudaSetupArgument* is intercepted, the frontend can hold them until *cudaLaunch* is triggered. This optimization reduces the number of interactions between the frontend and backend, which is the most significant overhead for small workload consolidation.

We now describe the analytical models the backend uses in making judicious decisions on consolidation.

V. GPU PERFORMANCE MODEL

We classify workload consolidation scenarios into two types, taking into consideration how thread blocks are distributed between SMs.

In the first type, each SM executes at most one thread block. For example, in encryption workload described above, thread blocks are distributed in round-robin fashion. Since each encryption instance occupies 3 SMs out of 30 SMs in Tesla C1060 GPU, consolidation of 6 instances occupies 18 SMs. Since each thread block goes to different SM, the thread blocks do not overlap with each other. For this type of consolidation, there is no need to consider how the thread blocks are scheduled within and across SMs and the performance model just needs to capture performance of single workload and the resource sharing effects of consolidation, in particular global memory bandwidth sharing.

In the second type of consolidation, more than one thread block are scheduled into each SM. In this case, thread blocks scheduled onto an SM either come from the same workload or from different workloads. For this type, the performance model must consider the GPU scheduling strategy across SMs, besides the concerns of resource sharing effects.

We extend a recent GPU performance model [8] to estimate the performance of consolidated workloads on GPU. The previous model designed for single workloads cannot directly be applied to predict consolidated kernel performance since the model is based on the implicit assumption that SMs are either idle or executing the same type of workload. This assumption is not true for the case of consolidation. In addition, the previous work does not consider the performance impact of thread blocks scheduling that is an important factor to determine performance of consolidated workloads. In the following, we will briefly review our model due to the space limitation, but a detailed model description can be found in [30].

For the first type of consolidation, we apply the existing model to each single kernel, but with a simple extension of considering global memory bandwidth sharing. We apply our model extension to two consolidated workloads of the first category. Figure 3 compare predicted performance with measured performance using our extended model. Our prediction captures potential performance variances due to workload consolidation and is accurate.

For the second type of consolidated workloads, different SMs may have different thread blocks. The SMs that finishes its workload latest determines the execution time. We call this SM as "critical SMs". To estimate the execution time of consolidated workloads, we first identify the critical SMs and then estimate the execution time of thread blocks scheduled on the critical SMs.

To identify the critical SMs, we need to know how the GPU schedules thread blocks to SMs. To explain why this matters to performance, we use the same scenarios shown in Tables 2 and 3 in Section III.

In both scenarios, the thread blocks are initially distributed between SMs in a round-robin fashion. In the first scenario (Table 2), since the encryption instance is much shorter than MC (Monte Carlo), it is possible that the first 15 SMs finish the encryption instance while the second 15 SMs are still working on the MC blocks. To balance workload between SMs, some untouched MC blocks will then be redistributed to the first 15 SMs by the GPU scheduler. Therefore the critical SMs, i.e. the first 15 SMs, are allocated with one encryption thread block plus 2 MC thread blocks. In the second scenario (Table 3), the warps from the BlackScholes thread blocks in the first 15 SMs are executed interleaving with the warps from the Search instance. Hence, the BlackScholes thread blocks in the first 15 SMs finish no earlier than any BlackScholes thread blocks in the second 15 SMs. Therefore, the critical SMs in this example, i.e first 15 SMs, are allocated with 1 Search block and 1 BlackScholes block.

In general, thread block distribution among SMs depends on execution time of a workload and the load balance principle of the GPU scheduler. Sometimes, the thread block re-distribution can happen in the middle of execution. We can determine critical SMs based on analyzing execution time of a workload and thread block distribution.

After identifying the thread blocks assigned to the critical SMs, we estimate execution time. SM schedules warps of thread blocks based on warp type, instruction type, and "fairness" to all warps executing in the SM [9]. To maximize system output, it is possible that SM interleaves different warps (i.e., the warps belonging to different workloads) to improve parallelism. We regard different types of workloads whose warps are scheduled into the same SM as one single *big workload* and estimate the execution time for this big workload.

We compare predicted performance with our model for the two scenarios with measured times (Figure 4) The prediction error is less than 12%. A prediction error is likely due to uneven distributed between SMs, where some SMs finish their work earlier and alleviate the memory bandwidth contention, while our model assumes bandwidth sharing always happens. We plan on investigating the reasons further.

VI. GPU POWER MODEL

In this section, we discuss predicting GPU Power for consolidated workloads. The GPU power model should capture execution properties of each workload that is being consolidated. The existing work in predicting GPU power [1] cannot be applied directly on consolidated workloads.

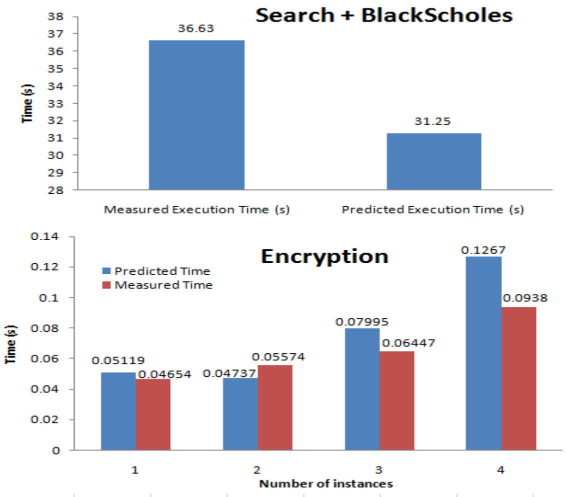


Figure 3: Execution time prediction for the first category of consolidated workloads (i.e, at most one thread block is allocated per SM)

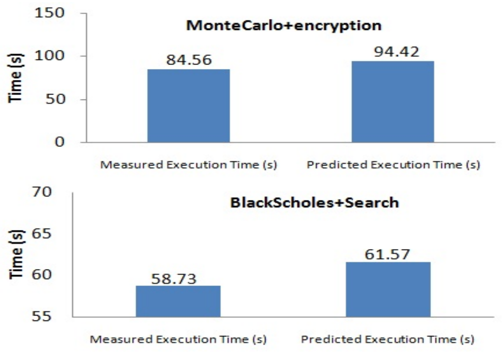


Figure 4: Execution time prediction for the second category of consolidated workloads (i.e., more than one thread blocks are allocated per SM)

Typically, power consumption of a GPU (P) is the sum of static power, dynamic power, and the impact of temperature:

$$P = P_{static} + P_T(\Delta T) + P_{dyn} \qquad (10)$$

P_{static} is static power, P_T is the temperature impact on power and P_{dyn} is dynamic power. Static power depends on chip layout and circuit technology, and is independent of workload execution. Chip temperature has an impact on power (P_T). The leakage current and thermal voltages for a transistor vary as temperature changes, which in turn leads to leakage power changes [3]. Dynamic power (P_{dyn}) results from transistors switching overhead.

Static power is measured when no workload is executed and while none of the GPU resources is turned off. The power consumed by leakage current and thermal voltages is measured using the linear relationship between various temperature changes (ΔT) as shown in [1]. We obtain the equation coefficients by running a set of training benchmarks. While the static and the thermal powers are independent of workload, dynamic power is dependent on actual usage of GPU hardware resources by a workload. The

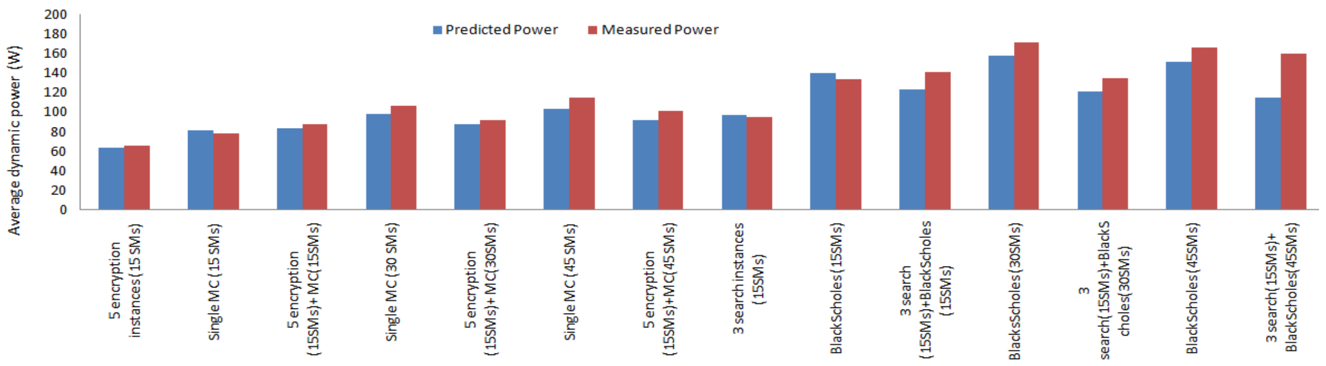


Figure 5: Comparison of predicted average power with measured average power

resources include floating point units, global memory, constant memory, shared memory, etc.

The dynamic power is the total power consumed by all GPU hardware components. If e_i is the event rate on component i, then the total dynamic power is:

$$P_{dyn} = \sum_{i=1}^{|s|} a_i e_i + \lambda \tag{11}$$

$$e_i = \frac{\textit{The number of occurrences of the event } i}{\textit{execution cycles}} \tag{12}$$

Event rate is the access rate of hardware component i in a single SM. e_i is the ratio of number of instructions that access a hardware component and the total number of execution cycles. The number of instructions that access a hardware component is calculated by analyzing PTX code that CUDA compiler generates. From our analysis, we observe that the two main components that contribute to the dynamic power the highest are: global memory accesses and computation instructions. We use the performance model described in the previous section for obtaining the total number of execution cycles and calculate the event rate.

The remaining model coefficients in Eq. 11 (α_i and λ) are obtained through empirical analysis while running a set of training benchmarks. In specific, we measure power and event rate (e_i) of each training benchmark and then derive the coefficients by performing linear regression. We train our model using 6 GPU benchmarks from Rodinia benchmark suite [7] (10 GPU kernels). To measure GPU power consumption, we first measure the whole system idle power (P_{idle}) that includes GPU static power (P_{static}). Then we measure the average system power (P_{sys}) when GPU is leveraged to run workloads. We fix cooling fans speed inside the machine to eliminate their power effects. We assume that when GPU is executing the workload, the other system components consume almost the same power as when the system is idle. Therefore, GPU power ($P_T(\Delta T)$ + P_{dyn}) can be estimated by ($P_{sys} - P_{idle}$). Our assumption is valid because the major power consumption contributors (CPU, host memory, fans, and disk) [14] use almost the same power as when the system is idle.

For heterogonous workload consolidation, different SMs may be executing different workloads, which leads to different event rates across SMs. The workload may also be distributed unevenly on SMs. Using the event rates from a specific SM cannot capture the power consumption of the other SMs. It is incorrect to simply estimate dynamic power for each SM based on their individual event rates and then add them up to obtain total dynamic GPU power. For instance, prediction error with such adding for the consolidation of encryption and MC is 9X times different from the actual measurement). That indicates power consumption estimated for one SM is in fact consumed by multiple SMs. To solve this power model problem, we assume a "virtual" SM whose event rates is the average event rates of all SMs. We estimate the power based on this virtual SM. We test our model with 14 variations of consolidated workloads. The prediction error of the GPU power model is less than 10% and achieve 6.4% on average (Figure 5).

VII. USAGE OF MODELS IN THE FRAMEWORK

We describe usage of our models to decide judicious workload consolidation for verifying energy efficiency at runtime. Figure 6 shows an overview of the decision making process. The backend keeps track of the number of workloads that issue GPU kernels. When the number reaches a certain threshold, the backend considers the workload consolidation; otherwise GPU kernels are executed as if there is no consolidation framework. We set the threshold to 10 times the number of available GPUs to represent a large number of GPU requests. This number can be adjusted based on further observation. When the number of workloads is large enough, the backend randomly chooses workload candidates according to the available consolidation templates. The backend then calculates performance and power by using the parameterized models described in Sections V and VI. The backend calculates energy consumption of the chosen consolidation kernel based on the number of chosen workload instances, and their parameters (i.e., numbers of thread blocks for each workload, number of computation instructions per thread, number of coalesced/uncoalesced memory type instructions per thread and total number of synchronization instructions per thread), and hardware architecture parameters (i.e., DRAM access latency, delay between two uncoalesced and

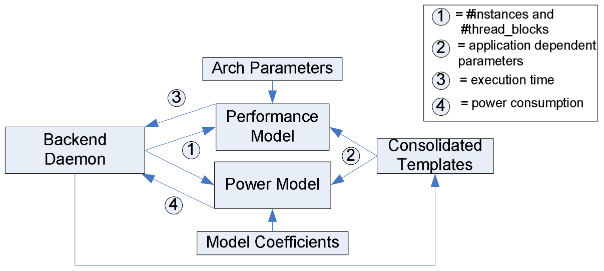


Figure 6: Integrating the models with the runtime systems

coalesced memory transactions, clock frequency of the SM processor, and bandwidth between the DRAM and cores). Most of the model parameters, except the number of instances and the number of threads blocks for each workload, are obtained offline. Hence, the overhead of calculating performance and energy benefits is low.

If the backend determines that consolidation of the chosen workloads is not beneficial, it lets the kernels run normally and chooses other workload candidates for consolidation. Since we assume to have enough workloads to schedule, it would be easier to overlap workload computation with energy benefit verification for reducing runtime overhead.

VIII. PERFORMANCE EVALUATION

We test our dynamic consolidation framework to verify throughput and energy consumption of various consolidated workloads. We execute various combinations of workloads listed in Table 1 on a machine with Intel Xeon E5520 quad-core CPU and Nvidia Tesla C1060 GPU. We compare GPU and CPU performance and energy consumption to demonstrate energy efficiency of consolidation. Furthermore, we investigate mitigation of consolidation framework overhead.

We measure the whole system power with the WattsUp? PRO ES power meter in the same way as described in [13]. For CPU power measurements, we turned off the GPU by disconnecting power to it. For GPU measurements, power consumption includes CPU power and GPU power. To measure the power for a workload whose execution time is small (less than 5 seconds), we run the workload multiple times and measure the average system power.

To evaluate the performance of consolidated workloads on CPU, we launch multiple workloads on the CPU concurrently and rely on the scheduling policy of OS (Ubuntu Server 9.10, Linux 2.6.31-22-server kernel) to distribute workloads among processor cores. To make a fair comparison, the CPU code is parallelized using OpenMP and is optimized for best memory accesses. In our implementations, most of CPU kernels perform better than GPU kernels.

We compare total execution time and total energy consumed for four execution setups. The first is running multiple instances on multicore CPU (labeled *CPU* in the

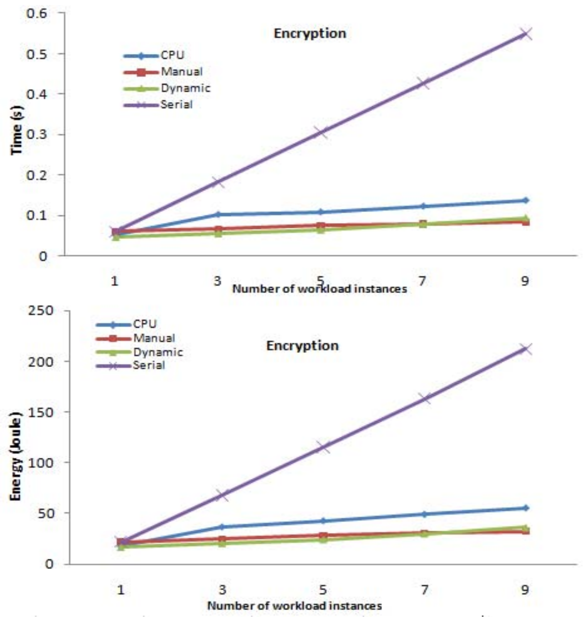


Figure 7: Results for running multiple instances of encryption workloads

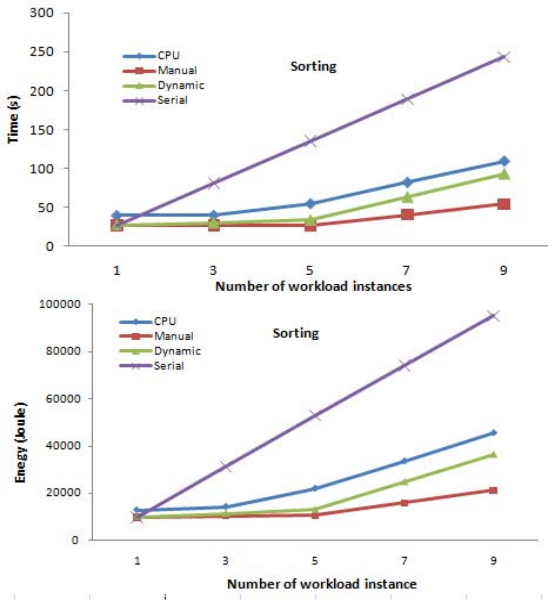


Figure 8: Results for running multiple instances of Sorting workloads

graphs). The next setup is running each instance of GPU workload serially without consolidation (labeled *serial*). This is the way current GPUs are typically used, where the CUDA scheduler issues one execution after another. *Manual* setup refers to manual implementation of consolidating instances of multiple instances. Although manual consolidation is not possible in a data center situation, we are simply using it as a reference since it does not include the overheads caused by intercepting user requests and extra data copying from different contexts into one buffer. The manual version also does not include the optimizations performed by our dynamic framework. Finally, *dynamic* refers to results with our runtime consolidation backend proposed in this paper.

Figure 7 shows two graphs, one with total execution time and the other with total energy consumption for executing various instances of the encryption workload. Each instance encrypts 12KB of data. We can see that serial execution on GPU has the worst performance in all cases as the execution of instances is serialized. Runtime consolidation leads to energy savings of up to 29% and performance benefit of up to 68% compared to CPU, although a single GPU instance performs worse in terms of both execution time and energy consumption. Optimizations such as reusing data required by multiple instances led to our consolidation framework's better performance than manual implementation when the number of instances is less than 3. Beyond that, data transfer overhead causes the performance to drop slightly. Even for these cases, runtime consolidation performs better than CPU. As we increase the number of instances to more than 9, the data transfer overheads become overwhelming and achieves neither high throughput nor energy savings. Note that the data transfer overheads can be estimated based on memory bandwidth and data size. Therefore, our framework in a real environment avoids choosing this case of too many instances for consolidation.

Figure 8 shows the results for various instances of sorting workload, each instance with 6K input elements. As we increase the number of instances, performance benefit of GPU with consolidation increases further from 1.4X to 2X (at 9 instances), compared to CPU. With serial execution, the GPU performance benefit is lost since multicore CPU can schedule multiple instances on different cores to fully utilize the hardware resources while GPU with serial execution cannot. GPU execution time with manual consolidation stays almost constant as we pack more instances of sorting workload on GPU, because the workloads are distributed evenly into separate SMs and do not introduce global memory access overhead. In other words, the GPU hardware utilization is improved without any performance loss. This is in contrast with the encryption workload where GPU execution time is slightly increased as we consolidate more instances. Our dynamic consolidation framework has similar performance as the manual consolidation up to 5 instances. As we increase the number of instances further, the overheads catch up and some performance is lost. Overall, runtime consolidation is still better than CPU for all numbers of instances. CPU execution time also increases significantly when the number of instances is larger than 4. As a result, CPU energy consumption starts to increase significantly.

Tables 5 and 6 show the total execution time and energy consumption for running multiple heterogeneous instances of Search workload and BlackScholes workload. In these tables, *x*S means *x* instances of Search workload and *y*B means *y* instances of BlackScholes workload. Similar

Table 5: Execution time (second) for running Search (S) and BlackScholes (B) workload instances

Benchmarks	CPU	Manual	Dynamic	Serial
1S+1B	60.3	36.6	38.1	69.4
1S+10B	218.4	37.4	40.2	377.2
2S+10B	220.5	38.1	41.1	412.5
1S+20B	401.7	38.4	43.4	719.2

Table 6: Total energy consumption (Joule) for running Search (S) and BlackScholes (B) workload instances

Benchmarks	CPU	Manual	Dynamic	Serial
1S+1B	24532.93	13572.59	14139.86	25730.32
1S+10B	95184.05	15061.74	16197.95	151902.1
2S+10B	89718.45	15568.41	16788.66	168271.2
1S+20B	176763.3	15736.89	17786.41	294683.6

Table 7: Execution time (second) for running Encryption (E) and MonteCarlo (M) workload instances

Benchmarks	CPU	Manual	Dynamic	Serial
1E+1M	387.7	57.2	57.2	88.9
3E+3M	605.5	57.4	57.5	266.8
4E+12M	976.6	57.7	57.8	701.5
5E+15M	1163.4	57.8	59.9	876.9

Table 8: Total energy consumption (Joule) for running Encryption (E) and MonteCarlo (M) workload instances

Benchmarks	CPU	Manual	Dynamic	Serial
1E+1M	162443	20617.81	20648.01	32058.44
3E+3M	263853.8	21697.55	21746.46	100838.4
4E+12M	427091.8	22309.35	22380.19	271439.5
5E+15M	511666.9	22451.37	23263.51	340546.2

notation is applied in describing the remaining heterogeneous workload consolidation results. Execution time of a single Search instance on CPU and GPU are 17 seconds and 35.2 seconds respectively, i.e., CPU performance is better. Execution time of a single BlackScholes instance on CPU and GPU are 57.4 seconds and 34.2 seconds, respectively, i.e., GPU performance is better. After we consolidate them together into one kernel on GPU, the consolidation effect leads to both performance and energy benefits on GPU compared to CPU. This result is interesting, since we can consolidate a workload that is performing well on GPU with a workload that is achieving worse performance on GPU than on CPU and improve their combined performance. In addition, we notice that (1S+20B) case, which has more workloads than the other cases, performs best in all cases (9.3X performance speedup and 9.9X energy savings, compared to CPU), which demonstrates the advantage of workload consolidation. We also notice that with our runtime workload consolidation, the performance is only slightly worse than with manual consolidation, due to small communication and data copy overhead incurred between the frontends and backend. Also, serial execution performance of these benchmarks for all cases is the worst as expected.

Tables 7 and 8 show the results for consolidating multiple heterogeneous instances of MonteCarlo workload and Encryption workload. Execution time of a single Encryption instance on CPU and GPU are 7.2 seconds and 45.7 seconds, respectively, i.e. CPU performance is better.

A single MonteCarlo instance finishes in 306 seconds on CPU and in 43.2 seconds on GPU, i.e. MonteCarlo's GPU performance is multiple times better than CPU. Serial execution on GPU gets the worst performance. Compared to CPU, consolidation on GPU manually and using our runtime framework performs significantly better. The overhead caused by our framework is negligible in this case. Workload consolidation achieves up to 19X performance speedup and 22X energy savings (using 5E+15M), compared to execution on CPU.

IX. CONCLUSIONS

This paper presents a case for energy-efficient use of GPUs in enterprise computing. We presented motivating examples to illustrate the benefits of energy-aware consolidation and to show the need for judicious consolidation. We have described power and performance prediction models for workload consolidation on GPUs and used them in our dynamic consolidation framework. We developed optimizations to reduce the overhead caused by our framework. For various benchmarks usage of our framework achieves energy savings in the range of 2X to 22X, and performance benefits in the same range. Despite upcoming technical advances in GPUs, our process-level consolidation is an energy efficient strategy and can complement future GPU architectures.

REFERENCES

[1] S. Hong and H. Kim, "*An Integrated GPU Power and Performance Model*," ISCA, 2010.

[2] S. Collange, D. Defour, and A. Tisserand, "*Power Consumption of GPUs from a Software Perspective*," Lecture nodes in Computer Science, 2009, Volumn 5544/2009, 914-923

[3] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "*Hotleakage: a temperature-aware model of subthreshold and gate leakage for architects*," Tech Report, University of Virignia, 2003

[4] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "*Temperature-aware microarchitecture: modeling and implementation,*" ACM Trans. Architecture. Code Optimization., 2004.

[5] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "*Statistical power modeling of GPU kernels using performance counters*," The first international green computing conference, 2010.

[6] *Cubin utilities—decuda*, http://wiki.github.com/laanwj/decuda.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S-H. Lee, and K. Skadron, "*Rodinia: a benchmark suite for heterogeneous computing*," IEEE International Symposium on Workload Characterization, 2009.

[8] S. Hong and H. Kim, "*An analytical model for a GPU architecture with memory-level and thread-level parallelism*," In proceedings of the 36th International Symposium on Computer Architecture, Austin, TX, 2009.

[9] E. Lindholm, J. Nickolls, S. Qberman, and J. Montrym. "*NVIDIA Tesla: a unified graphics and computing architecture*," IEEE Micro, Vol 28, issue 2, page 39-55, 2008.

[10] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. "*Enabling Task Parallelism in the CUDA Scheduler*," Workshop on Programming Models for Emerging Architectures, Sep. 2009.

[11] Federal Information Processing Standard, "*FIPS PUB 197: the official AES standard*," http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[12] D. Ren, R. Suda, "*Power Efficient Large Matrices Multiplication by Load Scheduling on Multi-core and GPU Platform with CUDA*," vol. 1, pp.424-429, International Conference on Computational Science and Engineering, 2009

[13] S. Huang, S. Xiao, and W. Feng, "*On the Energy Efficiency of Graphics Processing Units for Scientific Computing*," HP-PAC-2009.

[14] X. Feng, R. Ge, and K. W. Cameron, "*Power and Energy Profiling of Scientific Applications on Distributed Systems*," IPDPS '05, 2005.

[15] *cuobjdump*, Nvidia developer website, http://developer.nvidia.com/page/home.html

[16] *Fermi Compute Architecture White Paper*, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[17] S. Collange, D. Defour, and A. Tisserand, "*Power Consumpiton of GPU from a Software Perspective*," Lecture notes in computer science, volume 5544, page 914-923, 2009.

[18] H. Takizawa, K. Sato, and H. Kobayashi, "*SPRAT: Runtime Processor Selection for Energy-aware Computing*," IEEE International Conference on Cluster Computing, 2008.

[19] X. Ma, M. Dong, L. Zhong and Z. Deng, "*Statistical Power Consumption Analysis and Modeling for GPU-based computing*," Hotpower workshop, 2009.

[20] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, M. Sarrafzadeh, "*Energy-aware High Performance Computing with Graphic Processing Units*," Conference on power aware computing and systems, 2008.

[21] S. Ryoo, C. Rodrigues, S. Stone, and S. Baghsorkhi, S. Ueng, J. Stratton and W. W. Hwu, "*Prorgram Optimization Space Pruning for a Multithreaded GPU*," International Sysmposium on Code Generation and Optimization, 2008.

[22] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp and W. Hwu, "*An Adaptive Performance Modeling Tool for GPU Architecture*," In ACM PPoPP, 2010.

[23] A. Kerr, G. Diamos, and S. Yalamanchili, "*Modeling GPU-CPU Workloads and Systems*," In GPGPU workshop, 2010.

[24] W. Liu, W. Muller-Wittig, B. Schmidt, "*Performance Prediction for General-Purpose Computations on GPUs*," In ICPP, 2007.

[25] Nvidia CUDAZone, CUDA Community Showcase, http://www.nvidia.com/object/cuda_apps_flash_new.html

[26] Michael Kipper, Joshua Slavkin, Dmitry Denisenko, "Implementing AES on GPU", http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/AES_ON_GPU_report.pdf

[27] A. Kaatz, Parallel Sorting Speed competition, http://courses.ece.illinois.edu/ece498/al/HallOfFame.html

[28] Nvidia CUDA SDK Code Samples, http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

[29] T. Scogland, H. Lin, and W. Feng. "*A First Look at Integrated GPUs for Green High-Performance Computing*," International Conference on Energy-Aware High Performance Computing, 2010

[30] D. Li, S. Byna, and S. Chakradhar, "*Energy-Aware Workload Consolidation on GPU*", Virginia Tech technical report, 2011

[31] V. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "*Supporting GPU Sharing in Cloud Environment with a Transparent Runtime Consolidation Framework*", the International Symposium on High-Performance Parallel and Distributed Computing, 2011