Supplemental: Accurate and Reliable Energy Measurement and Modelling of Data Transfer between CPU and GPU in Parallel Applications on Heterogeneous Hybrid Platforms

Hafiz Adnan Niaz, *Member, IEEE*, Ravi Reddy Manumachu, *Member, IEEE*, and Alexey Lastovetsky, *Member, IEEE*

I. Introduction

THE supplementary material includes:

- Specifications of the two heterogeneous hybrid servers, *Haswell k40c GPU server* and *Icelake A40 GPU server*.
- Methodology to obtain ground-truth profiles of data transfer between computing devices in a heterogeneous hybrid platform (III).
- Energy predictive models of data transfer between computing devices in a heterogeneous hybrid platform (IV).
- Runtime software energy sensor API for computations and data transfers (V).
- Experimental results containing energy predictive models of computations for multicore CPUs, comparision of energies of computation and data transfers, and comparision of total energy using sensors versus ground-truth (VI).

II. Heterogeneous Hybrid Servers: Specifications

TABLE I: Specifications of the heterogeneous hybrid server containing a dual-socket Intel multicore Haswell CPU and a Nvidia K40c GPU. We name this server, *Haswell k40c GPU server*.

Intel Haswell E5-2670V3	
No. of cores per socket	12
Socket(s)	2
CPU MHz	1200.402
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
TDP	120 W
NVIDIA K40c GPU	
No. of processor cores	2880
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
TDP	245 W

H. Niaz, R. Manumachu, and A. Lastovetsky are with the School of Computer Science, University College Dublin (UCD), Belfield, Dublin 4, Ireland.
E-mail: hafiz.niaz@ucdconnect.ie, ravi.manumachu@ucd.ie, alexey.lastovetsky@ucd.ie

TABLE II: Specifications of the heterogeneous hybrid server containing a single-socket Intel Icelake multicore CPU and two Nvidia A40 GPUs. We name this server, *Icelake A40 GPU server*.

Intel Platinum 8362 Icelake	
No. of cores per socket	32
No. of threads per core	2
Socket(s)	2
L1d cache, L1i cache	1.5 MiB, 1 MiB
L2 cache, L3 cache	40 MiB, 48 MiB
Total main memory	62 GB DDR4-3200
TDP	265 W
NVIDIA A40 GPU	
No. of GPUs	2
No. of Ampere cores	10,752
Total board memory	48 GB GDDR6 (with ECC)
Memory bandwidth	696 GB/sec
TDP	300 W

III. Methodology to Obtain Ground-truth Profiles of Data Transfer Between CPU and GPU

A. Main Steps for Energy Measurement of Data Transfer Between a GPU and the Host CPU

The main steps to obtain the dynamic energy consumption for a data transfer of m bytes from a GPU to the host CPU are outlined below. *All the steps are invoked from the CPU side in the main thread.*

- We send the data of size m bytes from the host CPU to the GPU and keep this data on the GPU for the remainder of the measurement.
- Query the CPU processor clock to obtain the start time.
- Start the energy measurement on the CPU side using an energy measurement API's *start()* function [1].
- Pin the main thread to a CPU core dedicated to this data transfer using the system call, sched_setaffinity().
- Send the data of size m bytes from the GPU to the host CPU.
- Send a data item of size 1 byte from the host CPU to the GPU.
- Stop the energy measurement on the CPU side using the energy measurement API's *stop()* function [1].
- Query the CPU processor clock to get the end time and calculate the difference between the end and start times to get the execution time.

- The dynamic energy consumption of the data transfer is the difference between the total energy consumption and the static energy consumption, which is the static power consumption of the platform multiplied by the execution time of the data transfer.

B. Main Steps for Energy Measurement of p Data Transfers Between p Device Pairs

The main steps to obtain the dynamic energy consumption of parallel data transfers of data sizes, $\{m_1, \cdots, m_p\}$, between p pairs of devices where m_i is transferred between devices in the i-th pair are as follows:

- Query the CPU processor clock in the main thread to obtain the start time.
- Launch p pthreads from the program's main thread to perform the parallel data transfers.
- In each pthread, pin the thread to a dedicated CPU core using the system call, *sched_setaffinity()*.
- Start the energy measurement on the CPU side in the main thread using an energy measurement API's *start()* function [1].
- In pthread i ($i \in \{0, \cdots, p-1\}$), send the data of size m_i bytes from d_{i1} to d_{i2} in the i-th pair and a data item of size 1 byte from d_{i2} to d_{i1}.
- Invoke the *pthread_join()* call in the main thread to join with the data transfer pthreads.
- Stop the energy measurement on the CPU side in the main thread using the energy measurement API's *stop()* function [1].
- Query the CPU processor clock to get the end time and calculate the difference between the end and start times to get the execution time.
- The dynamic energy consumption of the p parallel data transfers is the difference between the total energy consumption and the static energy consumption, which is the static power consumption of the platform multiplied by the total execution time of the parallel data transfers.

Note that the data transfers in the pthreads and the start of energy measurement in the main thread are synchronized by a barrier, which aims to start these operations at the same time.

C. Statistical Methodology to Measure Execution Time and Energy Consumption

Consider a hybrid application, *app*, consisting of three data-parallel kernels, *Kernel_cpu*,*Kernel_gpu1*, and *Kernel_gpu2*, which run in parallel. The goal is to measure the execution time and the dynamic energy consumption of kernels in the application. To do this, we instrument the application, as shown in Algorithm 1. The instrumented application returns the execution time of each kernel and the energy consumption of all three kernels.

Consider a hybrid application, *app*, consisting of three data-parallel kernels, *Kernel_cpu*,*Kernel_gpu1*, and *Kernel_gpu2*, which run in parallel. The goal is to measure the execution time and the dynamic energy consumption of kernels in the application. To do this, we instrument the application, as shown in Algorithm 1. The instrumented application returns the execution time of each kernel and the energy consumption of all three kernels.

Algorithm 1 Instrumentation of a sample application (*app*) consisting of three kernels, executing on CPU and two GPUs simultaneously.

1: HCL_WATTSUP_START( )
2: #pragma parallel
3: **Begin**
4: $te_{cpu}1 \leftarrow gettimeofday()$
5: KERNEL_CPU( )
6: $te_{cpu}2 \leftarrow gettimeofday()$
7: **End**
8: **Begin**
9: $te_{gpu1}1 \leftarrow gettimeofday()$
10: KERNEL_GPU1( )
11: $te_{gpu1}2 \leftarrow gettimeofday()$
12: **End**
13: **Begin**
14: $te_{gpu2}1 \leftarrow gettimeofday()$
15: KERNEL_GPU2( )
16: $te_{gpu2}2 \leftarrow gettimeofday()$
17: **End**
18: $energy_{app} \leftarrow$ HCL_WATTSUP_STOP( )
19: $te_{cpu} \leftarrow te_{cpu}2 - te_{cpu}1$
20: $te_{gpu1} \leftarrow te_{gpu1}2 - te_{gpu1}1$
21: $te_{gpu2} \leftarrow te_{gpu2}2 - te_{gpu2}1$
22: **return** $(te_{cpu}, te_{gpu1}, te_{gpu2}, energy_{app})$

1) Methodology to Measure Execution Time: We instrument each kernel in the hybrid application (*app*) using the member function *gettimeofday()* of the Linux library *sys/time.h* to measure its execution time separately. As shown in Algorithm 1, the execution times are stored in variables te_{cpu}, te_{gpu1}, and te_{gpu2} and output at the end of the application execution.

2) Methodology to Measure the Energy Consumption: The heterogeneous hybrid node has one WattsUp Pro power meter that sits between the wall A/C outlets and the node's input power sockets. The power meters capture the total power consumption of the node. They have data cables connected to one USB port of the node. A Perl script collects the data from the power meter using the serial USB interface. The execution of these scripts is non-intrusive and consumes insignificant power.

The power meters are periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. The maximum sampling speed of the power meters is one sample every second. The accuracy specified in the data-sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts. The accuracy at 0.5 watts is ± 0.3 watts.

We use *HCLWattsUp* API [1], which gathers the readings from the power meters to determine the average power and energy consumption during the execution of an application for the whole node. *HCLWattsUp* API also provides two macros: *HCL_WATTSUP_START* and *HCL_WATTSUP_STOP*. The *HCL_WATTSUP_START* macro starts gathering power readings from the power meter using the Perl script, whereas the *HCL_WATTSUP_STOP* stops gathering and return the total energy as a sum of these power readings.

To measure the amount of energy consumed by the application, we invoke *HCL_WATTSUP_START* and *HCL_WATTSUP_STOP* macros, as shown in Algorithm

1. The consumed energy is stored in the variable, $energy_{app}$, that is output at the end of the application execution.

D. Methodology to Ensure Reliability of Experimental Results

Each application is instrumented for measuring its performance and energy consumption. We keep running the application until the sample averages of the measured execution times and energy consumption of the application lie within a given confidence interval, and a given precision is achieved. For this, we employ a script, which is named MEANUSINGTTEST. Algorithm 2 presents the pseudocode of this script. It executes the application *app* repeatedly until one of the following three conditions is satisfied:

1) The maximum number of repetitions ($maxReps$) is exceeded (Line 4).
2) The sample averages of all devices (kernel execution times and the application energy consumption) fall in the confidence interval (or the precision of measurement eps is achieved) (Lines 11-15).
3) The elapsed time of the repetitions of application execution exceeds the maximum allowed time ($maxT$ in seconds) (Lines 16-18).

MEANUSINGTTEST returns the sample averages of the execution times for each abstract processor (i.e. $time_{cpu}$, $time_{gpu1}$, $time_{gpu2}$) and the energy consumption of the kernels (i.e. $energy$). The input parameters are the minimum and maximum number of repetitions, $minReps$, and $maxReps$. These parameter values differ based on the workload size solved. For small workload sizes ($32 \leq n \leq 1024$), these values are set to 10000 and 100000. For medium workload sizes ($1024 < n \leq 5120$), these values are set to 100 and 1000. For large workload sizes ($n > 5120$), these values are set to 5 and 50. The values of $maxT$, cl, and eps are set to 3600, 0.95, and 0.1. If the precision of measurement is not achieved before the maximum number of repeats is exceeded, we increase the number of repetitions and the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in the experiments.

Algorithm 3 shows the pseudocode of the helper functions CALACCURACY, which is used by MEANUSINGTTEST. It returns one if the sample average of a given reading lies in the 95% confidence interval (cl), and a precision of 0.025 ($eps = 2.5\%$) is achieved. Otherwise, it returns 0.

If the precision of measurement is not achieved before the maximum number of repeats is exceeded, we increase the number of repetitions and the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in the experiments.

E. Steps to Prevent Noise in Measurements

Several steps are taken to measure energy consumption to eliminate any potential interference (noise) from components not involved in data transfer between the host CPU and a GPU.

Energy consumption of the data transfer between the host CPU and a GPU may include contributions from SSDs, NIC, and fans. Therefore, we ensure that these contributions are minimal using the steps below:

Algorithm 2 Script determining the mean of an experimental run using student's t-test.

```
1: procedure MEANUSINGTTEST(app, minReps, maxReps, maxT, cl, eps,
        reps#, elapsedTime, time_cpu, time_gpu1, time_gpu2, energy)
Input:
    The application to execute, app
    The minimum number of repetitions, minReps ∈ Z_{>0}
    The maximum number of repetitions, maxReps ∈ Z_{>0}
    The maximum time allowed for the application to run, maxT ∈ R_{>0}
    The required confidence level, cl ∈ R_{>0}
    The required accuracy, eps ∈ R_{>0}
Output:
    The number of experimental runs actually made, reps# ∈ Z_{>0}
    The elapsed time, elapsedTime ∈ R_{>0}
    The mean execution times, time_cpu, time_gpu1, time_gpu2 ∈ R_{≥0}
    The mean consumed energy, energy ∈ R_{>0}

2:     reps ← 0; stop ← 0; etime ← 0
3:     sum_cpu ← 0; sum_gpu1 ← 0; sum_gpu2 ← 0; sum_eng ← 0
4:     while (reps < maxReps) and (!stop) do
5:         (t_cpu[reps], t_gpu1[reps], t_gpu2[reps], eng[reps]) ← EXECUTE(app)

6:         sum_cpu += t_cpu[reps]
7:         sum_gpu1 += t_gpu1[reps]
8:         sum_gpu2 += t_gpu2[reps]
9:         sum_eng += eng[reps]
10:        if reps > minReps then
11:            stop_cpu ← CALACCURACY(cl, reps + 1, t_cpu, eps)
12:            stop_gpu1 ← CALACCURACY(cl, reps + 1, t_gpu1, eps)
13:            stop_gpu2 ← CALACCURACY(cl, reps + 1, t_gpu2, eps)
14:            stop_eng ← CALACCURACY(cl, reps + 1, t_eng, eps)
15:            stop ← stop_cpu ∧ stop_gpu1 ∧ stop_gpu2 ∧ stop_eng
16:            if max{sum_cpu, sum_gpu1, sum_gpu2} > maxT then
17:                stop ← 1
18:            end if
19:        end if
20:        reps ← reps + 1
21:    end while
22:    reps# ← reps
23:    elapsedTime ← max{sum_cpu, sum_gpu1, sum_gpu2}
24:    time_cpu ← sum_cpu / reps; time_gpu1 ← sum_gpu1 / reps; time_gpu2 ← sum_gpu2 / reps
25:    energy ← sum_eng / reps
26:    return (reps#, elapsedTime, time_cpu, time_gpu1, time_gpu2, energy)
27: end procedure
```

Algorithm 3 Algorithm Calculating Accuracy

```
1: function CALACCURACY(cl, reps, Array, eps)
2:     clOut ← fabs(gsl_cdf_tdist_Pinv(cl, reps − 1))
              × gsl_stats_sd(Array, 1, reps)
              / sqrt(reps)
3:     if clOut × reps / (Σ_{i=0}^{reps−1} Array[i]) < eps then
4:         return 1
5:     end if
6:     return 0
7: end function
```

- The program performing the data transfer between the host CPU and a GPU does not invoke any file I/O functions. We monitor the disk consumption during the program run and ensure that negligible I/O is performed by the program using tools such as *sar* and *iotop*.
- The sizes of the data transferred between the host CPU and a GPU do not exceed the CPU's and GPU's main memory. Therefore, we ensure that no swapping (paging) occurs during the program run on the CPU side and that there are no failures on the GPU side.
- The program performing the data transfer between the host CPU and a GPU does not invoke any network I/O functions. We monitor the network activity using tools such as *sar* and *atop* and ensure it is not significant.

Fans are significant contributors to energy consumption. Our

hybrid server platform controls fans in zones: a) zone 0: CPU or System fans, b) zone 1: Peripheral zone fans. There are four levels to control the speed of fans:

- Standard: Baseboard management controller (BMC) controls both fan zones, with the CPU and Peripheral zones set at speed 50%;
- Optimal: BMC sets the CPU zone at speed 30% and the Peripheral zone at 30%;
- Heavy IO: BMC sets the CPU zone at speed 50% and the Peripheral zone at 75%;
- Full: All fans running at 100%.

To rule out fans' contribution to dynamic energy consumption, we set the fans at full speed before running the programs. Therefore, the energy consumption by fans is included in the static power consumption of the server.

Furthermore, during the application run, we monitor the server's temperatures and the fans' speeds with the help of Intelligent Platform Management Interface (IPMI) sensors. We make sure that there are no changes in the temperatures and the fans' speeds.

F. Application of the Methodology for Two Heterogeneous Hybrid Servers

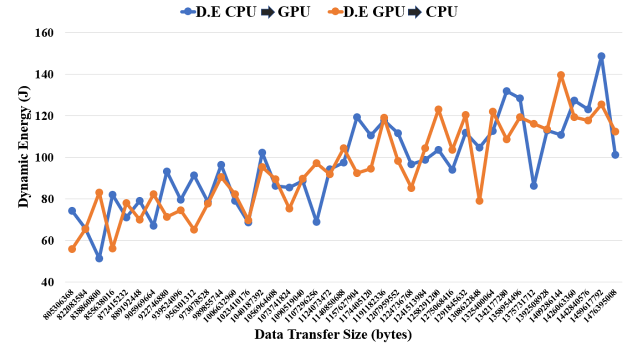
We apply our methodology to obtain the ground-truth dynamic energy profiles of data transfers between devices ((CPU, K40c GPU) in heterogeneous server shown in Table I) and between pairs of devices (CPU, A40 GPU_1) and (CPU, A40 GPU_2) in the heterogeneous server shown in Table II.

We analyze 1) the dynamic energy and execution time of data transfers in forward and backward directions between the host CPU and a GPU, 2) the dynamic energy and execution time of data transfers between pairs of devices in serial, 3) the dynamic energy and execution time of parallel data transfers between pairs of devices, and 4) The execution times involved in obtaining the profiles.

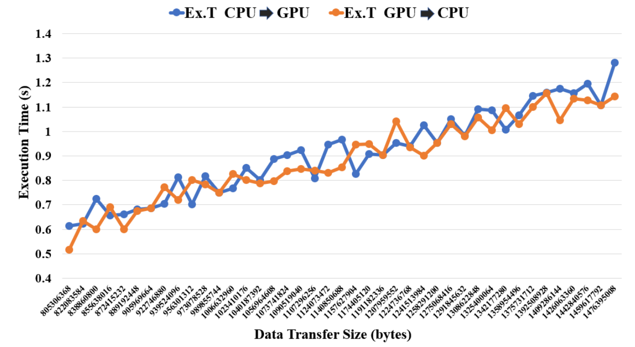
Figure 1 shows the dynamic energy and execution time profiles of data transfers between a pair of computing devices (CPU, K40c GPU) and (K40c GPU, CPU) in the heterogeneous server (Table I). Figure 1a shows the dynamic energy and execution time of data transfer between the host CPU and K40c GPU. Similarly, Figure 1b depicts the case for the data transfer between the K40c GPU and host CPU.

Figures 2a and 2b show the dynamic energy consumption and execution time of data transfer between the pairs of computing devices, (CPU, A40 GPU_1) and (A40 GPU_1, CPU), respectively, in the Icelake A40 GPU server. Similarly, the Figures 2c and 2d depict the case for the pair of devices, (CPU, A40 GPU_2) and (A40 GPU_2,CPU).

Figures 3a and 3b show the dynamic energy consumption and execution time of data transfers happening in parallel between two pairs of devices, (CPU, A40 GPU_1) and (CPU, A40 GPU_2). Figure 3c shows the sum of the dynamic energy consumptions of data transfer between the pair of computing devices (CPU, A40 GPU_1) followed by data transfer between the pair (CPU, A40 GPU_2), and data transfers between the two pairs of devices in parallel. Figure 3d shows the sum of the dynamic energy consumptions of data transfer between the



(a) Dynamic Energy of data transfer from CPU $\Longrightarrow$ K40c GPU, and K40c GPU $\Longrightarrow$ CPU.



(b) Execution time of data transfer from CPU $\Longrightarrow$ K40c GPU, and K40c GPU $\Longrightarrow$ CPU.

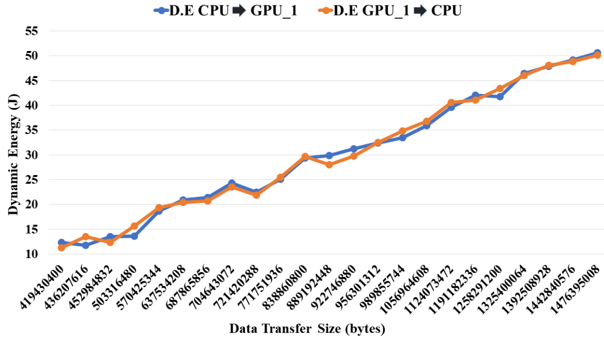
Fig. 1: Dynamic energy and execution time profiles of data transfers between the pairs of computing devices (CPU,K40c GPU) and (K40c GPU,CPU) in the Haswell k40c GPU server.

pair of devices (A40 GPU_1, CPU) followed by data transfer between the pair (A40 GPU_2, CPU), and data transfers between the two pairs of devices in parallel.

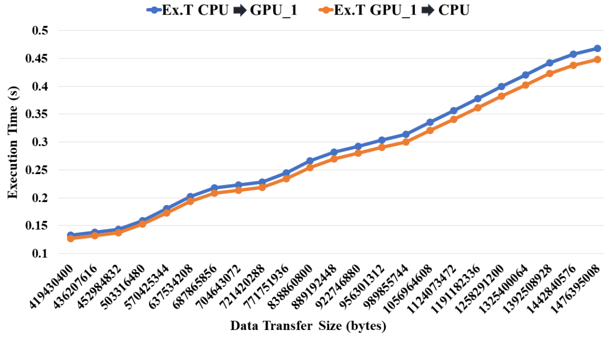
IV. Energy Predictive Models of Data Transfer Between Computing Devices

In this work, we propose a fast selection procedure that combines a natural grouping of performance events derived from the processor architecture, additivity, the theory of energy predictive models of computing and high positive correlation to select a small subset of performance events that can be employed in linear energy models to accurately predict the energy of data transfer between a host CPU and a GPU.

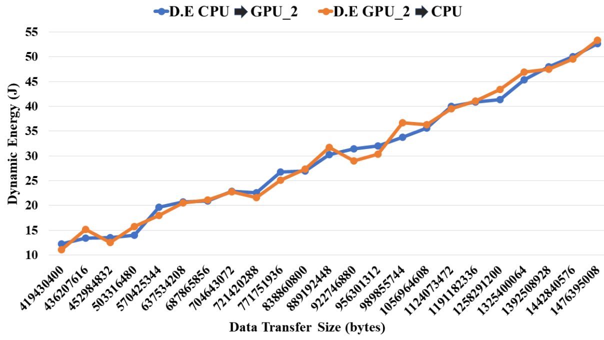
Likwid [2] offers 16 counters and more than 1500 performance events for the Intel Haswell multicore CPU in the Haswell k40c GPU server. It offers 347 counters and 2772 performance events for the Intel IceLake multicore CPU in the Icelake A40 GPU server. One can obtain 16 and 347 performance event counts in one application run on these platforms. PMC (Performance Monitoring Counters) is one such counter group that captures core-level performance events. Likwid provides eight counters in PMC for the Intel IceLake multicore CPU processor. These eight counters can store 258 core-level performance events. Therefore, one gathers eight event counts in 8 PMCs in one application run. To gather



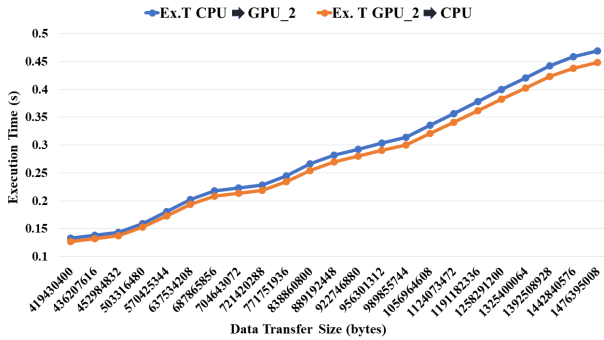
(a) Dynamic energy of data transfer from CPU $\Longrightarrow$ A40 GPU_1 and A40 GPU_1 $\Longrightarrow$ CPU.



(b) Execution time of data transfer from CPU $\Longrightarrow$ A40 GPU_1 and A40 GPU_1 $\Longrightarrow$ CPU.



(c) Dynamic energy of data transfer from CPU $\Longrightarrow$ A40 GPU_2 and A40 GPU_2 $\Longrightarrow$ CPU.



(d) Execution time of data transfer from CPU $\Longrightarrow$ A40 GPU_2 and A40 GPU_2 $\Longrightarrow$ CPU.

Fig. 2: Dynamic energy and execution time profiles of data transfers between the pairs of computing devices, (CPU, A40 GPU_1), (A40 GPU_1, CPU), (CPU, A40 GPU_2) and (A40 GPU_2, CPU), in the Icelake A40 GPU server.

all the 258 core-level performance events, one must run the application 33 times.

The CUDA Profiling Tools Interface (CUPTI) [3] tool provides performance events and metrics for Nvidia A40 GPU that are typically employed for performance profiling. Like the performance events for multicore CPUs, the CUPTI events and metrics have also been used in dynamic energy predictive models [4], [5], [6]. For the Nvidia A40 GPU, the Nsight Compute profiler is used to obtain the events and the metrics. However, they are also significant in number. Furthermore, the number of events increases with each new processor generation.

A. *Likwid Performance Monitoring Counter Groups for the Multicore CPUs*

TABLE III: Performance monitoring counter groups in Intel IceLake multicore CPU of the Haswell k40c GPU server.

Performance Counter Groups	Description
BBOX	Measurements of the Home Agent (HA) in the uncore
CBOX	Last level cache (LLC) coherency engine in the uncore
MBOX	Integrated memory controllers (iMC) in the uncore
PBOX	Measurements of the Ring-to-PCIe (R2PCIe) interface in the uncore
PMC	Core-local general purpose counters
PWR	Measurements of the current energy consumption through the RAPL interface
QBOX	Measurements of the QPI Link layer (QPI) in the uncore
SBOX	Socket internal traffic through ring-based networks
UBOX	System configuration controller, interrupt traffic, and system lock master
WBOX	Power control unit (PCU) in the uncore

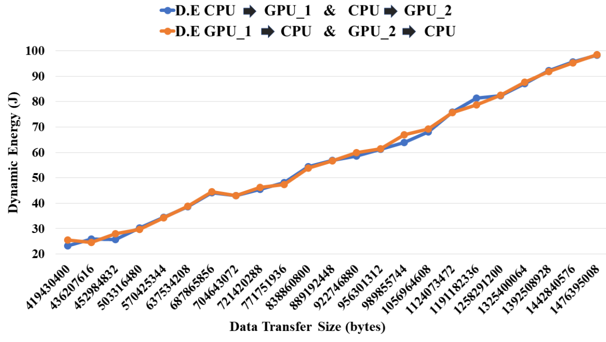
TABLE IV: Performance monitoring counter groups in Intel IceLake multicore CPU of the Icelake A40 GPU server.

Performance Counter Groups	Description
CBOX	Last level cache (LLC) coherency engine in the uncore
IBOX	Responsible for maintaining coherency for Integrated Input/Output controller (IIO) traffic
MBOX	Integrated memory controllers (iMC) in the uncore
M2M	Mesh2Mem (M2M) which connects the cores with the Uncore devices. The interface between the Mesh and the Memory Controllers.
PMC	Core-local general purpose counters
PWR	Measurements of the current energy consumption through the RAPL interface
QBOX	Measurements of the QPI Link layer (QPI) in the uncore
SBOX	Socket internal traffic through ring-based networks
TCBOX	Integrated Input/Output controller (IIO) counters
UBOX	System configuration controller, interrupt traffic, and system lock master
WBOX	Power control unit (PCU) in the uncore

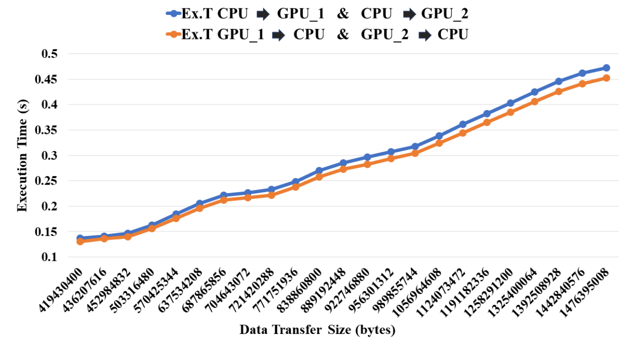
B. *Shortlisted Performance Events*

The procedure has three stages and is automated in a software library (**libedm**) that we developed for this purpose [7].

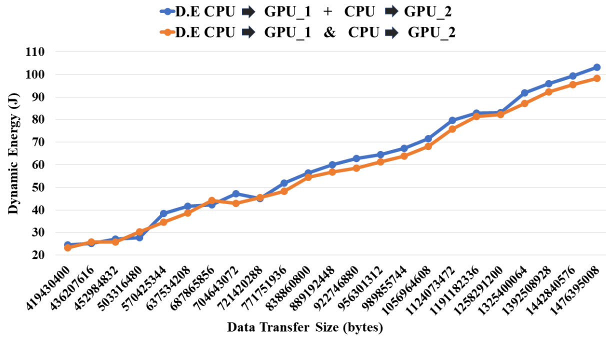
In the first stage, all highly additive (with additivity error $\leq$ 5%) and highly positively correlated performance events with dynamic energy ($\geq$ 95%) are selected.



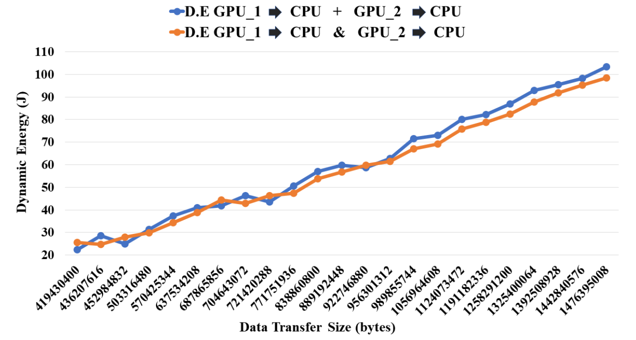
(a) Dynamic energy of data transfers between two pairs of devices CPU $\Longrightarrow$ A40 GPU_1 & CPU $\Longrightarrow$ A40 GPU_2 in parallel, A40 GPU_1 $\Longrightarrow$ CPU & A40 GPU_2 $\Longrightarrow$ CPU in parallel.



(b) Execution time of data transfers between two pairs of devices CPU $\Longrightarrow$ A40 GPU_1 & CPU $\Longrightarrow$ A40 GPU_2 in parallel, A40 GPU_1 $\Longrightarrow$ CPU & A40 GPU_2 $\Longrightarrow$ CPU in parallel.



(c) Comparison between the sum of serial dynamic energies from CPU $\Longrightarrow$ A40 GPU_1 + CPU $\Longrightarrow$ A40 GPU_2, and dynamic energy of data transfers between two pairs of devices in parallel CPU $\Longrightarrow$ A40 GPU_1 & CPU $\Longrightarrow$ A40 GPU_2.



(d) Comparison between the sum of serial dynamic energies from A40 GPU_1 $\Longrightarrow$ CPU + A40 GPU_2 $\Longrightarrow$ CPU, and dynamic energy of data transfers between two pairs of devices in parallel A40 GPU_1 $\Longrightarrow$ CPU & A40 GPU_2 $\Longrightarrow$ CPU.

Fig. 3: Dynamic energy and execution time profiles of data transfers happening in parallel between pairs of devices, (CPU, A40 GPU_1), (CPU, A40 GPU_2), in the Icelake A40 GPU server.

The main steps of the first stage that determines the highly additive and positively correlated performance events are as follows:

- Query the CPU processor clock to obtain the start time. Start the energy meter on the CPU side using an energy measurement API's *start()* function [1].
- Execute a base or compound application in the input dataset.
- Stop the energy meter on the CPU side using the energy measurement API's *stop()* function [1].
- Query the CPU processor clock to get the end time and calculate the difference between the end and start times to get the elapsed execution time. The dynamic energy consumption of the application execution is obtained.
- The event values during the data transfers in the application are obtained using the *Likwid-perfctr* tool.
- We verify that the execution time and dynamic energy consumption of a compound application is the sum of the execution times and dynamic energy consumptions of the constituent base applications.
- For each application, create an event record containing the dynamic energy and performance event values. The event records are then normalized.
- The performance events with insignificant counts (less than or equal to 10) and not deterministic and reproducible are eliminated. A performance event is deemed deterministic and reproducible if it exhibits the same value (within a tolerance of 5.0%) for different executions of the same application with the same runtime configuration on the same platform.
- The additivity error of a performance event for a compound application is calculated as follows: Additivity error (%) = $|\frac{(e_A+e_B)-e_{AB}}{(e_A+e_B+e_{AB})/2}| \times 100$ where e_{AB}, e_A, e_B are the performance event counts for the compound (AB) and constituent base applications, A and B, respectively. The sample average of this error is obtained from multiple experimental runs. The additivity error of the performance event is the maximum of errors for all the compound applications in the dataset.
- The performance events with additivity error ($\leq \mathbb{A}$) are selected.
- Finally, from the selected highly additive performance events, performance events with positive correlation ($\geq \rho$) are shortlisted for the second stage.

In the second stage, we determine the final shortlist of performance events, starting with a natural grouping of performance events derived from the processor architecture. This stage comprises two main steps: the *intra-group* and *inter-group* steps. The intra-group step prunes the performance events in each group based on correlation with each other and dynamic energy consumption. The inter-group step prunes the performance events across groups based on correlation with each other.

In the second stage, one can employ two approaches to finding a grouping of performance events that are highly correlated with each other and selecting one performance event as a representative from a group. The first approach is a *bottom-up approach (cluster by synthesis)* where all pair-wise correlations are analyzed to create meaningful clusters. However, this approach is practically infeasible since the number of performance events is significant.

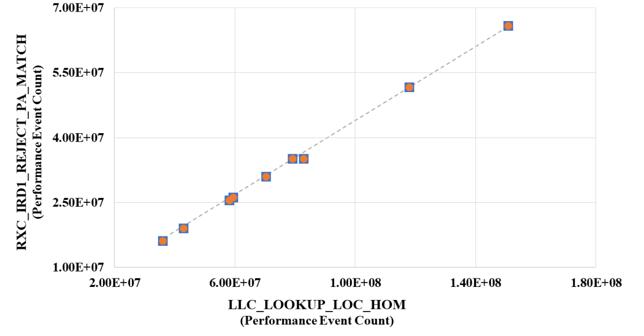
The second approach is a *top-down approach (cluster by analysis)*, which begins with groups of performance events that naturally exist together since they record the activities of the same hardware component in processor architecture. For example, Table IV in the Supplemental shows 11 performance counter groups, {CBOX, ..., WBOX}, provided by Likwid for the Intel Icelake multicore CPU in the Icelake A40 GPU server. The CBOX performance monitoring counter group contains the performance events occurring in the last level cache (LLC) coherency engine in the uncore. The WBOX group has performance events recording the activities of the power control unit (PCU) in the uncore. The correlations of the performance events in the groups are then analyzed individually (intra-group) and collectively (inter-group), leading to a final shortlist of performance events. We follow this approach in this paper since Likwid provides such an architecture grouping of performance events for all the multicore CPU platforms employed in this work. However, this approach is challenging for an arbitrary platform with no such natural grouping.

Consider the division of the M2M group into multiple sub-groups in the intra-group step of the second stage. Figures 4b and 4d show a high positive correlation between the events of one sub-group and the lack of it between the events of two different sub-groups of the M2M group. Similarly, Figures 4a and 4c show the sub-groups for CBOX group. Furthermore, Figures 5a and 5b show the high positive correlation of the representatives of sub-groups in CBOX and M2M.

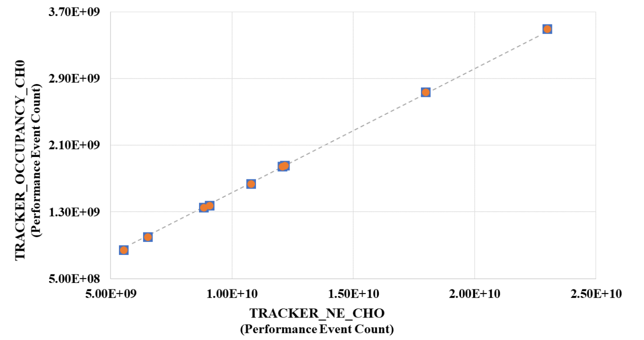
Figure 6 shows the correlation between representatives of sub-groups in different performance monitoring counter groups. For example, the first figure CBOX:M2M at the top refers to the correlation plot between representative performance events of sub-groups, TOR_OCCUPANCY_EVICT and TXR_HORZ_CYCLES_NE_BL_CRD, in CBOX and M2M, respectively.

C. Training and Testing of Energy Predictive Models of Data Transfer

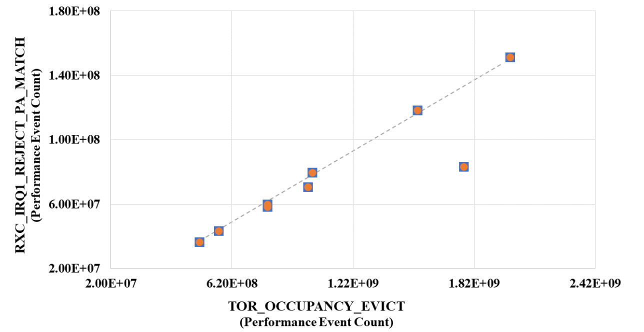
Figures 7, 8, and 9 compare the dynamic energy of serial and parallel data transfers predicted by the models against the



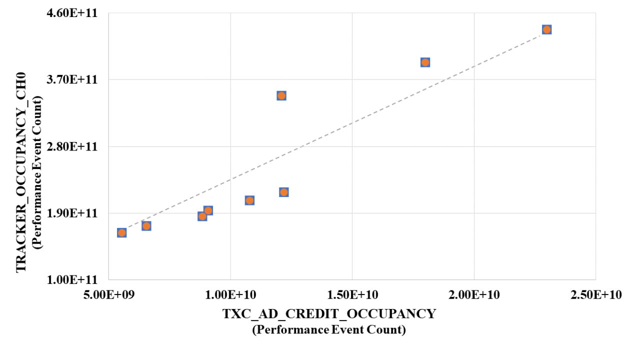
(a) Correlation between performance events of one sub-group in the group CBOX.



(b) Correlation between performance events of one sub-group in the group M2M.

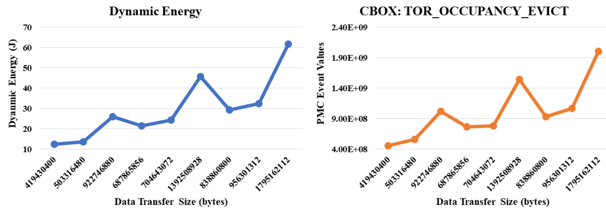


(c) Correlation between performance events of two different sub-groups in the group CBOX.

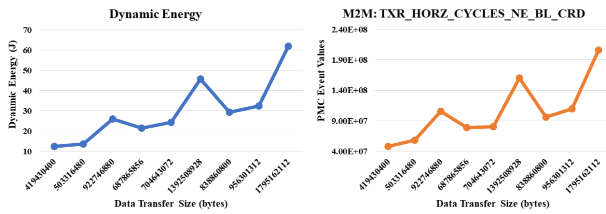


(d) Correlation between performance events of two different sub-groups in the group M2M.

Fig. 4: The correlation between the performance events of one sub-group and two different sub-groups within the CBOX and M2M groups.



(a) Correlation of CBOX performance event with data transfer dynamic energy.



(b) Correlation of M2M performance event with data transfer dynamic energy.

Fig. 5: Correlation of the performance events with dynamic energy in the groups, CBOX and M2M. The performance events are highly positively correlated with dynamic energy and follow the same trend as the dynamic energy.

TABLE V: Platform-level energy predictive models based on performance events for both directions for the Haswell k40c GPU server.

CPU $\Longrightarrow$ K40c GPU		
Group Name	**Performance Events**	**Model Coefficients**
MBOX	WR_CAS_RANK1_BANK14	4.69E-09
PMC	L2_RQSTS_RFO_HIT	7.45E-08
CBOX	TOR_INSERTS_NID_AL	1.78E-07
BBOX	RING_AK_USED_CW_EVEN	3.84E-07
PWR	PWR_PKG_ENERGY	1.81E-10
RBOX	RXR_INSERTS_DRS	2.26E-07

K40c GPU $\Longrightarrow$ CPU		
Group Name	**Performance Events**	**Model Coefficients**
MBOX	RD_CAS_RANK0_BANK7	4.69E-09
PMC	L2_TRANS_ALL_REQUESTS	7.45E-08
CBOX	TOR_INSERTS_WB	1.78E-07
BBOX	RING_AK_USED_CW_ODD	3.84E-07
PWR	PWR_DRAM_ENERGY	1.81E-10
RBOX	RXR_INSERTS_DRS	2.26E-07
UBOXFIX	UNCORE_CLOCK	1.44E-09

TABLE VI: Platform-level energy predictive models based on performance events for both directions for the Icelake A40 GPU server. Note there is only one performance event in each cell in the column “Performance Events”. Some performance events are wrapped to the next line due to space constraints.

CPU $\Longrightarrow$ A40 GPU_1		
Group Name	**Performance Events**	**Model Coefficients**
PMC	OFFCORE_REQUESTS_OUTSTANDING _DEMAND_RFO	3.27E-09
SBOX	VN1_NO_CREDITS_NCB	4.59E-10
TCBOX	TXN_REQ_OF_CPU_MEM _WRITE_IOMMU1	4.53E-09

A40 GPU_1 $\Longrightarrow$ CPU		
Group Name	**Performance Events**	**Model Coefficients**
M2M	TXR_HORZ_CYCLES_NE_BL_CRD	9.1E-10
PMC	OFFCORE_REQUESTS_OUTSTANDING_CYCLES _WITH_DEMAND_RFO	1.91E-08
SBOX	VN0_NO_CREDITS_SNP	1.84E-09
UBOX	M2U_MISC2_TXC_CYCLES_EMPTY_AKC	2.06E-11
WBOX	PKG_RESIDENCY_C0_CYCLES	2.3E-08

CPU $\Longrightarrow$ A40 GPU_1, CPU $\Longrightarrow$ A40 GPU_2 in Parallel		
Group Name	**Performance Events**	**Model Coefficients**
M2M	TRACKER_OCCUPANCY_CH0	5.64E-09
CBOX	LLC_LOOKUP_LOC_HOM	8.14E-10
SBOX	VN1_NO_CREDITS_NCB	5.1E-10
WBOX	PKG_RESIDENCY_C0_CYCLES	3.55E-08

A40 GPU_1 $\Longrightarrow$ CPU, A40 GPU_2 $\Longrightarrow$ CPU in Parallel		
Group Name	**Performance Events**	**Model Coefficients**
M2M	TXR_HORZ_CYCLES_NE_BL_CRD	1.55E-08
CBOX	TOR_OCCUPANCY_EVICT	9.36E-11
SBOX	VN0_NO_CREDITS_SNP	9.36E-11
TCBOX	REQ_FROM_PCIE_CMPL_IOMMU_HIT	6.58E-09
UBOX	M2U_MISC2_TXC_CYCLES_EMPTY_AKC	1.18E-08

ground-truth approach employing power measurements using physical external power meters for the heterogeneous server platforms (Tables I and II).

The summary of our findings are presented below:

- The energy predictive models for the two data transfer directions in the Haswell k40c GPU server share a performance event, which disallows the development of independent runtime software sensors based on the models for dynamic energy prediction of parallel data transfers in the two directions.
- The energy predictive models employ disjoint sets of performance events for the two data transfer directions in the Icelake A40 GPU server. However, the models give the same dynamic energy prediction for the same data transfer message size, complementing that the ground-truth dynamic energy profiles are almost the same for both directions for this server. This finding signifies that different performance events (belonging to the same performance monitoring counter groups) are able to comprehensively capture the energy consumption activities of data transfers in the two directions. Therefore, on the one hand, while having a large number of performance events (per performance monitoring counter group and

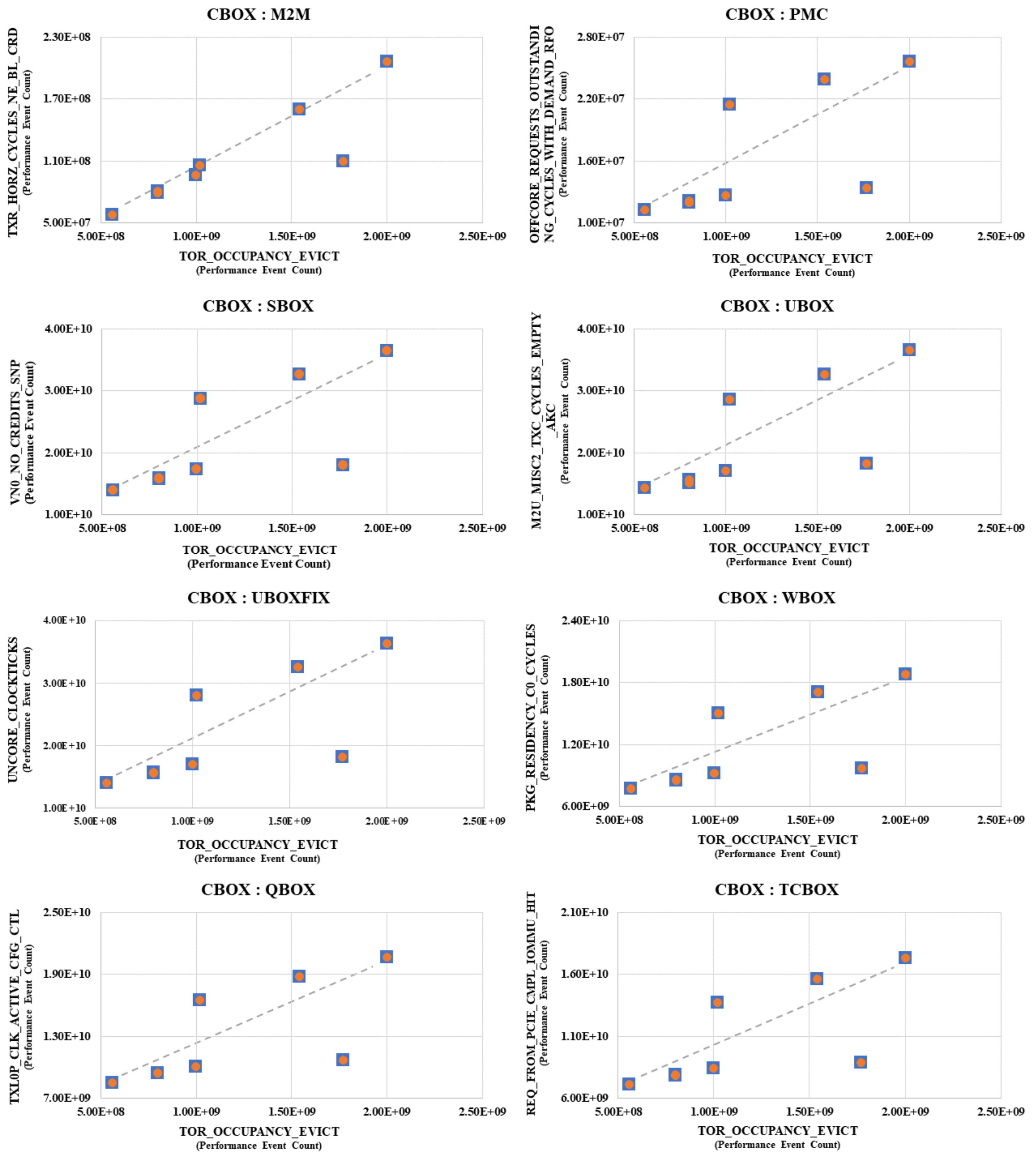
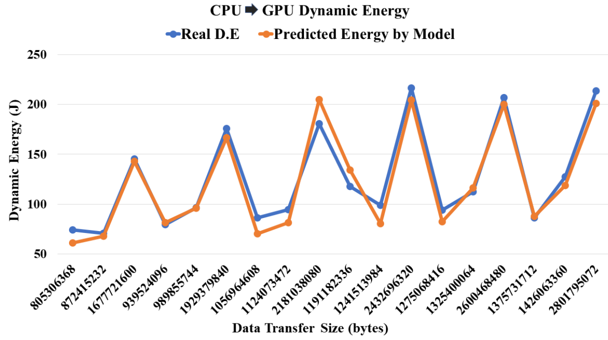
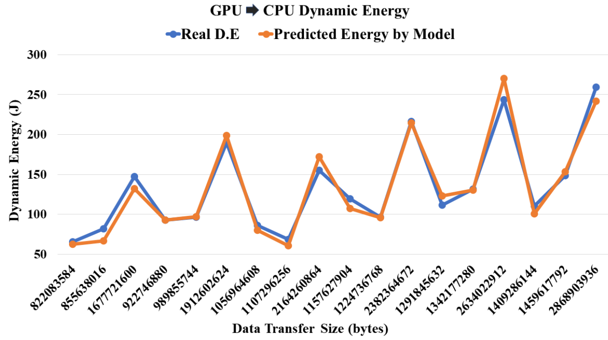


Fig. 6: Correlation between representative performance events of sub-groups in different performance monitoring counter groups. For example, the first figure CBOX:M2M at the top refers to the correlation plot between representative performance events of sub-groups, TOR_OCCUPANCY_EVICT and TXR_HORZ_CYCLES_NE_BL_CRD, in CBOX and M2M, respectively.

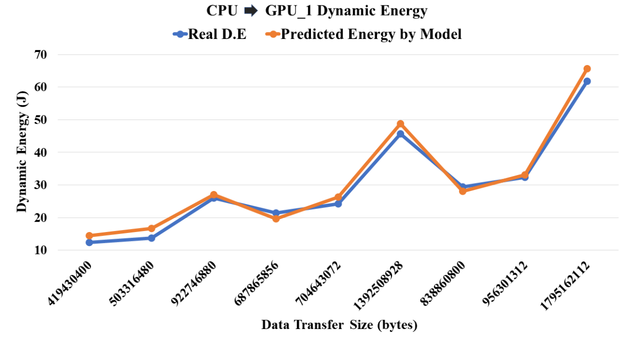


(a) Dynamic energy comparison of data transfers between pair of devices (CPU $\Longrightarrow$ K40c GPU) measured by ground-truth measurement method with the energy predicted by model based on performance events.

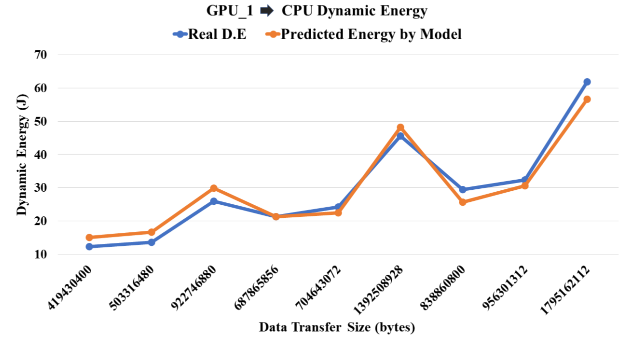


(b) Dynamic energy comparison of data transfers between pair of devices (K40c GPU $\Longrightarrow$ CPU) measured by ground-truth measurement method with the energy predicted by model based on performance events.

Fig. 7: Dynamic energy comparison of data transfers between pair of devices (CPU,K40c GPU) and (K40c GPU,CPU) measured by ground-truth measurement method with the energy predicted by model based on performance events for the heterogeneous hybrid server as shown in Table I.



(a) Dynamic energy comparison of data transfers between pair of devices CPU $\Longrightarrow$ A40 GPU_1 measured by ground-truth measurement method with the energy predicted by model based on performance events.



(b) Dynamic energy comparison of data transfers between pair of devices A40 GPU_1 $\Longrightarrow$ CPU measured by ground-truth measurement method with the energy predicted by model based on performance events.

Fig. 8: Dynamic energy comparison of data transfers between pair of devices (CPU,A40 GPU_1) and (A40 GPU_1,CPU) measured by ground-truth measurement method with the energy predicted by model based on performance events for the heterogeneous hybrid server shown in Table II.

overall) complicates the selection of an optimal subset of events to use for modelling, on the other hand, it affords an excellent opportunity to find disjoint sets of performance events that allows developing independent runtime software sensors based on models employing the disjoint sets for dynamic energy prediction of serial or parallel data transfers in the two directions.

- The energy predictive models for parallel data transfers between the host CPU and the two A40 GPUs employ disjoint sets of performance events. Notably, the difference between the dynamic energy for a parallel data transfer obtained using the model for parallel data transfers and the sum of the dynamic energies obtained using the direction-specific individual data transfer models is negligible. This means that a data transfer energy model for a direction can be used to obtain the dynamic energy for the parallel data transfers in that direction. This finding paves the way for developing a data transfer sensor for a direction, based on the data transfer energy model, to estimate the dynamic energy of parallel data transfers for that direction.

D. Energy Predictive Models of Computations for Multicore CPUs

Table VII shows the list of applications employed for training and testing the energy predictive models of computations for the Intel IceLake multicore CPU.

Table VIII shows the model variables employed in the linear energy predictive model of computations on Intel IceLake multicore CPU. The model variables are obtained using our methodology for shortlisting the performance events employing additivity and correlation.

There is no overlap in the performance events in the data transfer energy predictive models and component-level energy predictive models of computations for the hybrid applications executed on our research platforms. The data transfer energy predictive models contain performance events that capture off-core chip traffic whereas the energy predictive models of computations mainly employ the core-level performance events belonging to PMC counter group. This finding allows

TABLE VII: List of benchmarks in the application suite employed for training and testing the energy predictive models of computations for the Intel IceLake multicore CPU.

Application	Description
MKL FFT	Intel optimized 2-dimensional fast Fourier transform
MKL DGEMM	Intel optimized dense matrix multiplication of two square matrices
HPCG	Intel optimized High Performance Conjugate Gradient. 3-dimensional regular 27-point discretization of an elliptic partial differential equation
NPB IS	Integer Sort, Kernel for random memory access that sort small integers using the bucket sort technique
NPB LU	Lower-Upper Gauss-Seidel solver
NPB EP	Embarrassingly Parallel random number generator
NPB BT	Solve synthetic system of nonlinear partial differential equations using Block Tri-diagonal solver
NPB MG	Approximate 3-dimensional discrete Poisson equation using the V-cycle Multi Grid on a sequence of meshes
NPB FT	A 3D fast Fourier Transform partial differential equation benchmark
NPB DC	Arithmetic Data Cube, a data intensive grid benchmark representing data mining operations
NPB UA	Unstructured Adaptive mesh solving heat equation with convection and diffusion from moving ball
NPB CG	Solving an unstructured sparse linear system using Conjugate Gradient method
NPB SP	Solve synthetic system of nonlinear partial differential equations using Scalar Penta-diagonal solver
NPB DT	A graph benchmark evaluating communication throughput (Data Traffic)
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
HPGMG	High Performance Geometric Multigrid
miniFE	Unstructured implicit finite element codes
miniMD	Parallel molecular dynamics (MD) code
Naive MM	Naive Matrix-matrix multiplication
Naive MV	Naive Matrix-vector multiplication

TABLE VIII: Performance events used in the linear energy predictive model of computations on Intel IceLake multicore CPU.

Groups	Performance Events	Model Coefficients
TCBOX	IOMMU1_PWT_CACHE_LOOKUPS	2.24E-08
QBOX	TXL0P_CLK_ACTIVE_DFX	4.69E-09
UBOXFIX	UNCORE_CLOCKTICKS	3.11E-09

predicting accurately the component-level energy consumption of a hybrid application comprising parallel overlapping computations and data transfers that are crucial for the bi-objective optimization of the application for performance and energy.

Figures 10a, 10b, and 10c compares the dynamic energy of computations obtained by employing our energy predictive computation models versus the ground-truth dynamic energy profiles for the the three hybrid applications.

V. Runtime Software Energy Sensor API for Computations and Data Transfers

A. Software Energy Sensor API

libedm provide software energy sensor API functions that allow the creation of software energy sensors based on our proposed linear energy predictive models to estimate the energy consumption of computations and data transfers.

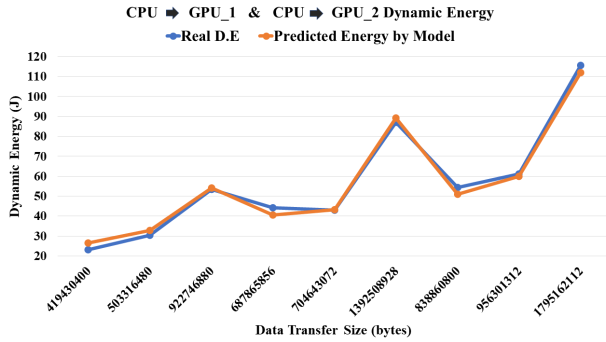
The API function *hcl_create_energy_csensor()* creates a software energy sensor for computations on a device represented by a tuple, $\{dtype, devno\}$, where *dtype* represents the device type (CPU or GPU) and *devno* signifies the device number within a class of devices given by the device type. The function returns a handle to the energy sensor in *sensor* on successful sensor creation. The argument *peventnames* is an array of size *npe* containing the names of the performance events. The argument, *modelf*, is a user-defined model function, which takes as input an array of performance event values and returns the dynamic energy consumption of the computations. For example, we pass a model function in the sensor creation call that estimates the dynamic energy consumption using our linear energy predictive models based on the performance event values provided as input.

```
typedef double ( *hcl_energy_model_func )(
  int npe, const double* peventarray);
int hcl_create_energy_csensor(
  hcl_device_type dtype, int devno,
  int npe, const char** peventnames,
  const hcl_energy_model_func* modelf,
  hcl_energy_sensor* sensor
);
```

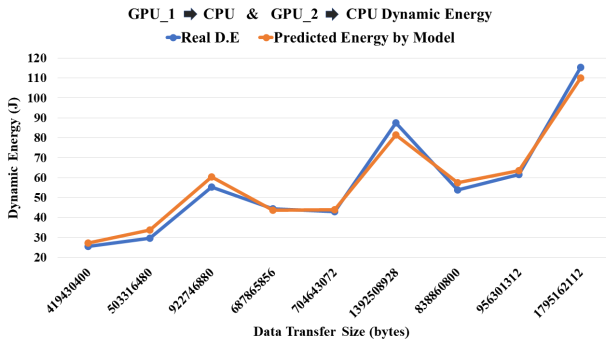
The API function *hcl_create_energy_dsensor()* creates a software energy sensor for data transfers between the host CPU and one or more GPUs in the direction given by the *direction* parameter. The direction parameter accepts two values, *FORWARD* and *BACKWARD*, representing the data transfers from the host CPU to the GPUs and GPUs to the host CPU, respectively. The argument, *modelf*, is a user-defined model function that returns the dynamic energy consumption of the data transfers given the performance event values. On successfully creating the sensor, the output argument, *sensor*, contains a handle for the energy sensor. Note that only one sensor can be employed for each direction to estimate the dynamic energy consumption of data transfers happening in that direction in a code region.

```
int
hcl_create_energy_dsensor(
  int direction, int npe,
  const char** peventnames,
  const hcl_energy_model_func* modelf,
  hcl_energy_sensor* sensor
);
```

The API functions *hcl_start_energy_sensor()* and *hcl_stop_energy_sensor()* start and stop the sensor represented by the identifier *sensor* for energy estimation.



(a) Dynamic energy comparison of data transfers between pair of devices (CPU,A40 GPU_1) & CPU $\Longrightarrow$ A40 GPU_2 measured by ground-truth measurement method with the energy predicted by model based on performance events for parallel data transfers.



(b) Dynamic energy comparison of data transfers between pair of devices (A40 GPU_1 $\Longrightarrow$ CPU) and (A40 GPU_2 $\Longrightarrow$ CPU) measured by ground-truth measurement method with the energy predicted by model based on performance events for parallel data transfers.

Fig. 9: Dynamic energy comparison of parallel data transfers between pairs of computing devices (CPU,A40 GPU_1), (CPU,A40 GPU_2) and (A40 GPU_1,CPU), (A40 GPU_2,CPU) using ground-truth approach and energy predictive models based on performance events for the Icelake A40 GPU server.

```
int
hcl_start_energy_sensor(
  const hcl_energy_sensor* sensor);
int
hcl_stop_energy_sensor(
  const hcl_energy_sensor* sensor,
  double *estenergy);
```

To estimate the energy consumption of a code region containing computations, the API function *hcl_start_energy_sensor()* is called before the start of the region, and the API function *hcl_stop_energy_sensor()* is called after the end of the region. However, some constraints exist on employing the software energy sensors for data transfer in a heterogeneous hybrid application. The software energy sensors for data transfer must be started in the main thread or an OpenMP master thread in a parallel region before initiating the data transfers and stopped in the main thread or an OpenMP master thread after the transfer of results from the GPUs to the host CPU. In the following section, we will illustrate the usage of data transfer sensors in a matrix multiplication application.

The API function *hcl_destroy_energy_sensor()* below releases the resources associated with the software energy sensor, *sensor*.

```
int
hcl_destroy_energy_sensor(
  hcl_energy_sensor* sensor);
```

B. Illustration of the Software Energy Sensor API in a Parallel Matrix Multiplication Application

We illustrate using the software energy sensor API functions through a parallel matrix multiplication application executing on our Icelake A40 GPU server.

Figure 13b illustrates the hybrid parallel matrix multiplication application (HDGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$.

The application has three software components: CPU, A40 GPU_1, and A40 GPU_2. All the components share the matrix B. The matrices A and C are horizontally partitioned such that each software component is assigned several contiguous rows of A and C provided as an input parameter to the application. The CPU software component is assigned N_1 rows of A and C. The A40 GPU_1 software component is assigned N_2 rows of A and C. Finally, the A40 GPU_1 software component is assigned the remaining $(N - N_1 - N_2)$ rows of A and C. Each software component *i* ($i \in \{1, 2, 3\}$) computes the matrix product, $C_i = A_i \times B$.

The application comprises three main stages. The first stage consists of data transfers of A_2, B, and C_2 from the host CPU to A40 GPU_1 and A_3, B, and C_3 from the host CPU to A40 GPU_2. The second stage involves local computations in the software components. The computations in the CPU software component are performed using the Intel MKL DGEMM library routine. The computations in the software components involving the A40 GPUs are performed using the CUBLAS DGEMM library routine. The third stage involves data transfer of the result matrices C_2 and C_3 from A40 GPU_2 and A40 GPU_1 to the host CPU.

Figures 11 and 12 illustrate the use of our proposed energy sensor API functions to estimate the energy consumption of computations and data transfers.

The application-specific routines, *cpudgemm()*, *gpudgemm1()*, and *gpudgemm2()*, contain the device-specific computations. The routines, *gpudt1()* and *gpudt2()*, comprise the data transfers between the host CPU and A40 GPU_1 and the host CPU and A40 GPU_2.

We first describe the code shown in Figure 11. Lines 4-28 contain the arrays of shortlisted performance events for computations on the multicore CPU and data transfers from CPU to A40 GPU_1 and A40 GPU_1 to CPU.

Line 29 contain the declarations of the software energy sensors. Lines 30-35 contain the API invocations creating the software energy sensors for computations on the CPU and two

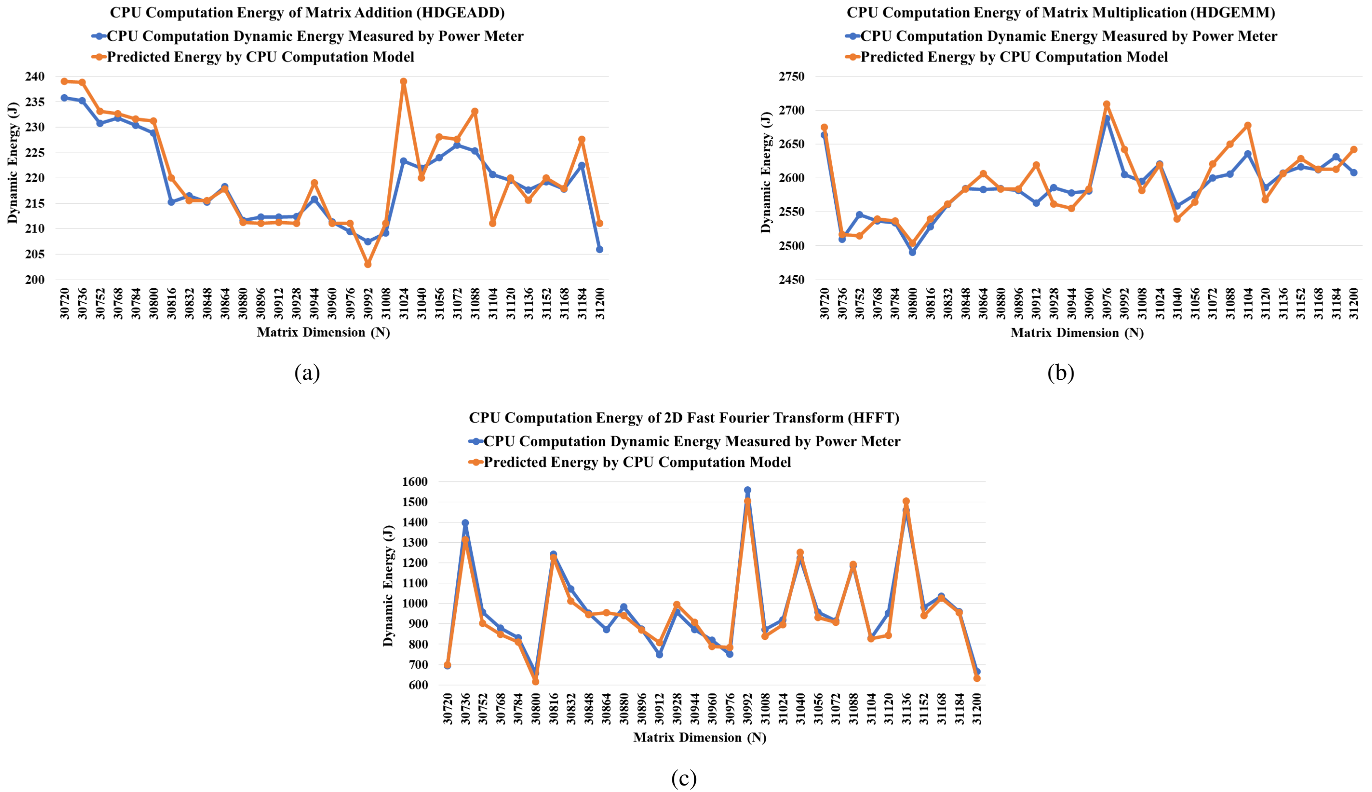


Fig. 10: Ground-truth dynamic energy profiles versus the dynamic energy of computations estimated by energy predictive models of computation for the three hybrid applications. (a). HDGEADD, (b). HDGEMM, and (c) HFFT.

GPUs. The sensor, $csensor[0]$, estimates the energy consumption of computations on the CPU. The sensors, $csensor[1]$ and $csensor[2]$ estimate the energy consumption of computations on the A40 GPU_1 and A40 GPU_2.

Lines 36-39 contain the API function invocations to create the data transfer sensors. The sensors, $cpu2gpusensor$ and $gpu2cpusensor$, estimate the energy consumption of data transfers from the host CPU to the A40 GPUs and the A40 GPUs to the host CPU, respectively. Note that the sensors for data transfers are different in the two directions: host CPU to GPU and GPU to host CPU.

The model function $cpumodelf$ estimates the energy consumption of computations given the values of the performance events in the array, *cpucevents*. The pre-defined model function hcl_nvml_modelf is based on NVML and estimates the energy of computations on the A40 GPUs. The model functions, $cpu2gpumodelf$ and $gpu2cpumodelf$, estimate the energy consumption of data transfers from the host CPU to the GPUs and GPUs to the host CPU given the values of the performance events in the array, *cpu2gpuevents* and *gpu2cpuevents*, respectively.

Figure 12 illustrates the rest of the code. Line 3 starts the parallel OpenMP region comprising three OpenMP threads that execute the three software components in parallel.

Line 3 comprises the parallel OpenMP region that launches three OpenMP threads to execute the three software components in parallel. In Lines 8-9, we invoke the API function, $hcl_start_energy_sensor()$, to start data transfer sensors, $dsensor[0]$ and $dsensor[1]$. Lines 13-15 comprise the CPU software component execution. In Lines 13 and 15, we call the API functions, $hcl_start_energy_sensor()$ and $hcl_stop_energy_sensor()$, to start and stop the sensor $csensor[0]$ for estimating the dynamic energy consumption of computations on the CPU. The estimated energy consumption is returned in the variable, $cpuenergy$.

Lines 18-21 contain the code for the A40 GPU_1 software component execution. In Lines 19 and 21, we invoke the API functions, $hcl_start_energy_sensor()$ and $hcl_stop_energy_sensor()$, to start and stop the sensor $csensor[1]$ for estimating the dynamic energy consumption of computations on the A40 GPU_1. Similarly, Lines 23-26 contain the A40 GPU_2 software component execution and API functions employing a sensor to estimate the dynamic energy consumption of computations on the A40 GPU_2. The estimated energy consumptions of computations on the GPU are returned in the variables $gpu1energy$ and $gpu2energy$.

In Lines 31-32, we invoke the API function, $hcl_stop_energy_sensor()$, to stop the data transfer sensors, $dsensor[0]$ and $dsensor[1]$. The estimated dynamic energy consumption of the data transfers from the CPU to A40 GPUs and the A40 GPUs to the CPU are returned in the variables $cpu2gpuenergy$ and $gpu2cpuenergy$, respectively. Note the data transfer of results from the A40 GPU_1 to host CPU happens in *gpudgemm1()* that invokes CUBLAS DGEMM routine and from the A40 GPU_2 to host CPU in *gpudgemm2()*. The barrier in Line 11 ensures that the data

```
#include <edm.h>
int main(int argc, char *argv[]) {
   double *A, *B, *C;
   int ncpuevents = 6; ncpu2gpuevents = 8,
       ngpu2cpuevents = 10;
   char* cpucevents[] = {
      "CORE_POWER_LVL0_TURBO_LICENSE",
      "TXC_AD_CREDIT_OCCUPANCY,"
      "IOMMU1_PWT_CACHE_LOOKUPS",
      "VN0_NO_CREDITS_SNP",
      "TXL0P_CLK_ACTIVE_DFX",
      "CBOX_CLOCKTICKS" };
   char* cpu2gpuevents[] = {
   "TRACKER_OCCUPANCY_CH0",
   "OFFCORE_REQUESTS_OUTSTANDING_DEMAND_RFO",
   "LLC_LOOKUP_LOC_HOM",
   "UPI_CLOCKTICKS",
   "VN1_NO_CREDITS_NCB",
   "TXN_REQ_OF_CPU_MEM_WRITE_IOMMU1" };
   char* gpu2cpuevents[] = {
   "TXR_HORZ_CYCLES_NE_BL_CRD",
   "OFFCORE_REQUESTS_OUTSTANDING_CYCLES_WITH_DEMA
   "TOR_OCCUPANCY_EVICT",
   "TXL0P_CLK_ACTIVE_CFG_CTL",
   "VN0_NO_CREDITS_SNP",
   "REQ_FROM_PCIE_CMPL_IOMMU_HIT",
   "UNCORE_CLOCKTICKS",
   "M2U_MISC2_TXC_CYCLES_EMPTY_AKC",
   "PKG_RESIDENCY_C0_CYCLES"};
   hcl_energy_sensor csensor[3], dsensor[2];
   hcl_create_energy_csensor(HCL_CPU, 0,
     ncpuevents, cpuevents, cpumodelf,
         &csensor[0]);
   hcl_create_energy_csensor(HCL_GPU, 0,
     0, NULL, hcl_nvml_modelf, &csensor[1]);
   hcl_create_energy_csensor(HCL_GPU, 1,
     0, NULL, hcl_nvml_modelf, &csensor[2]);
   hcl_create_energy_dsensor(FORWARD,
      ncpu2gpuevents, cpu2gpuevents,
     cpu2gpumodelf, &dsensor[0]);
   hcl_create_energy_dsensor(BACKWARD,
      ngpu2cpuevents, gpu2cpuevents,
     gpu2cpumodelf, &dsensor[1]);
   /* More to follow */
}
```

Fig. 11: Illustration of the libedm's software energy sensor API in a hybrid parallel matrix multiplication application executing on a server comprising a Intel multicore CPU and two Nvidia A40 GPUs. Five energy sensors are deployed in this application. Three sensors for computations on the CPU, A40 GPU_1, and A40 GPU_2.

transfer sensors are started successfully before initiating the data transfers from the host CPU to the GPUs. Finally, the barrier in Line 29 ensures that the data transfers from the GPUs to the host CPU are complete before stopping the sensors to get the dynamic energy estimate.

Lines 35-40 contain the API function invocations to destroy the sensors involved in estimating the energy of the application's computations and data transfers.

```
#include <edm.h>
int main(int argc, char *argv[]) {
#pragma omp parallel sections num_threads(3)
{
#pragma omp parallel num_threads(3)
{
#pragma omp master {
   hcl_start_energy_sensor(&dsensor[0]);
   hcl_start_energy_sensor(&dsensor[1]);
}
#pragma omp barrier
   if (omp_get_thread_num() == 0) {
     hcl_start_energy_sensor(&csensor[0]);
     cpudgemm(N1, N, N, A, B, C);
     hcl_stop_energy_sensor(&csensor[0],
         &cpuenergy);
   } else {
     if (omp_get_thread_num() == 1) {
       gpudt1(N2, N, N, &A[N1*N], B,
           &C[N1*N]);
       hcl_start_energy_sensor(&csensor[1]);
       gpudgemm1(N2, N, N);
       hcl_stop_energy_sensor(&csensor[1],
           &gpu1energy);
     } else {
       gpudt2(N1+N2, N, N, &A[(N1+N2)*N], B,
           &C[(N1+N2)*N]);
       hcl_start_energy_sensor(&csensor[2]);
       gpudgemm2(N1+N2, N, N);
       hcl_stop_energy_sensor(&csensor[2],
           &gpu2energy);
     }
   }
#pragma omp barrier
#pragma omp master {
   hcl_stop_energy_sensor(&dsensor[0],
       &cpu2gpuenergy);
   hcl_stop_energy_sensor(&dsensor[1],
       &gpu2cpuenergy);
}
}
   hcl_destroy_energy_sensor(&csensor[0]);
   hcl_destroy_energy_sensor(&csensor[1]);
   hcl_destroy_energy_sensor(&csensor[2]);
   hcl_destroy_energy_sensor(&dsensor[0]);
   hcl_destroy_energy_sensor(&dsensor[1]);
   exit(EXIT_SUCCESS);
}
```

Fig. 12: The matrix multiplication application contains three OpenMP threads invoking the energy sensor API functions and software components (kernels) in parallel, one CPU component, and two GPU components. Five energy sensors are deployed in this application: three for computations in the CPU, A40 GPU_1, and A40 GPU_2 software components ($csensor[0]$, $csensor[1]$, and $csensor[2]$), two for data transfers between the host CPU and the GPUs and the GPUs and the host CPU ($cpu2gpusensor$, $gpu2cpusensor$).

VI. Experimental Results

We experimentally determine the accuracy of our software energy sensors using three highly optimized parallel scientific applications on our Icelake A40 GPU server. The three heterogeneous hybrid applications are Matrix Addition (HDGEADD), Matrix Multiplication (HDGEMM), and 2D fast Fourier Transform (2D-HFFT). We employ the same matrix sizes for all the applications, starting from 30720x30720 to 31200x31200 with a stepsize of 16. For each matrix size, we determine the following:

- Estimated and actual dynamic energy of data transfers between pairs of devices by employing software energy sensors and the ground-truth method.
- Estimated and actual dynamic energy of computations on the multicore CPU by employing the software energy sensor and the ground-truth method.
- Estimated and actual dynamic energy of computations on the two GPUs using the software energy sensors and the ground-truth method.
- The total dynamic energy consumption during the application execution using the ground-truth method.
- The sum of the estimated dynamic energy consumptions of computations and data transfers during the application execution using our software energy sensors.

A. Comparison between Energies of Computations and Data Transfers

Figure 14 shows the dynamic energy consumption of computations and data transfers for various matrix sizes employed in three parallel applications, matrix addition, matrix multiplication and 2D fast Fourier transform, executed on a heterogeneous hybrid platform shown in Table II. The dynamic energy consumption is measured using the ground-truth method.

The dynamic energy of data transfer is significant compared to the computations on the multicore CPU processor for all three applications, contravening the widespread notion that data transfers consume negligible dynamic energy. The averages of the ratios of dynamic energy of computations to data transfer for the three applications are equal to 2, 5, and 2.5, respectively. Furthermore, the dynamic energy relationship with matrix size is non-linear for matrix addition, suggesting an opportunity for energy optimization.

Therefore, optimizing the data transfers in the parallel applications on heterogeneous hybrid platforms for performance and energy is an important problem.

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under the SFI Frontiers for the Future Programme 20/FFP-P/8683.

References

[1] M. Fahad and R. R. Manumachu, "HCLWattsUp: Energy API using system-level physical power measurements provided by power meters," 2022. [Online]. Available: https://csgitlab.ucd.ie/manumachu/hclwattsup

[2] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th international conference on parallel processing workshops*. IEEE, 2010, pp. 207–216.

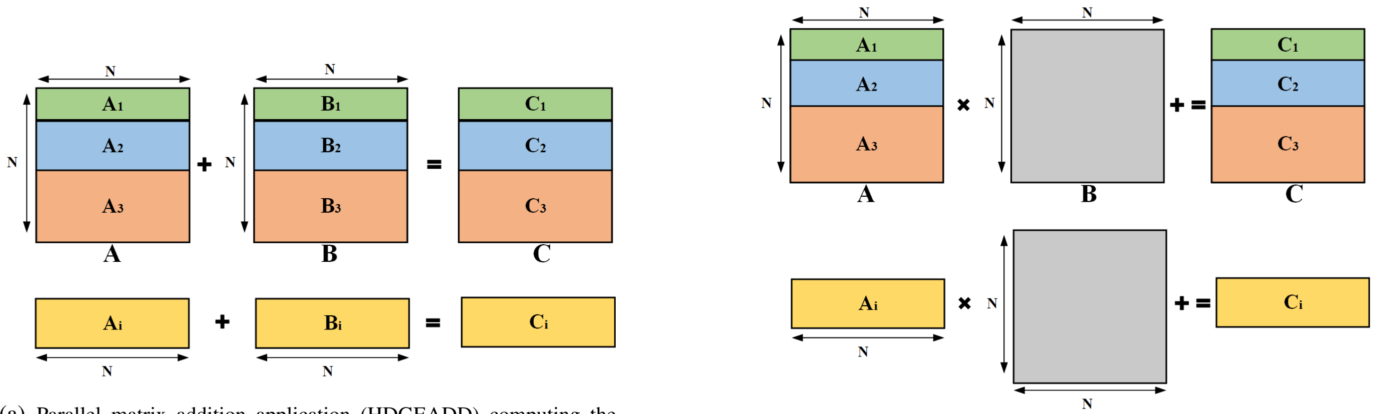
[3] NVIDIA Corporation, "CUDA profiling tools interface (CUPTI) - 1.0," 2021. [Online]. Available: https://developer.nvidia.com/cupti-1_0

[4] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in *International Green Computing Conference and Workshops (IGCC)*. IEEE, 2010.

[5] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: A statistical approach," in *Proceedings of the IEEE Sixth International Conference on Networking, Architecture, and Storage*, ser. NAS '11. IEEE Computer Society, 2011, pp. 149–158.

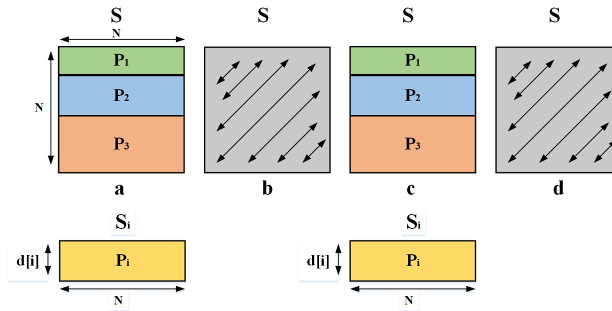
[6] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2013, pp. 673–686.

[7] Heterogeneous Computing Laboratory (HCL), "Energy modeling of software components in parallel applications on heterogeneous hybrid platforms," https://csgitlab.ucd.ie/manumachu/libedm.git, 2024.



(a) Parallel matrix addition application (HDGEADD) computing the matrix addition ($C = A + B$) of two dense square matrices A and B of size $N \times N$.The application comprises 3 software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each software component *i* computes the matrix addition, $C_i = A_i + B_i$.

(b) Matrix partitioning between the software components in hybrid parallel matrix multiplication application (HDGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$. The matrix B is shared by all the components. The application comprises 3 software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each software component *i* computes the matrix product, $C_i = A_i \times B$.

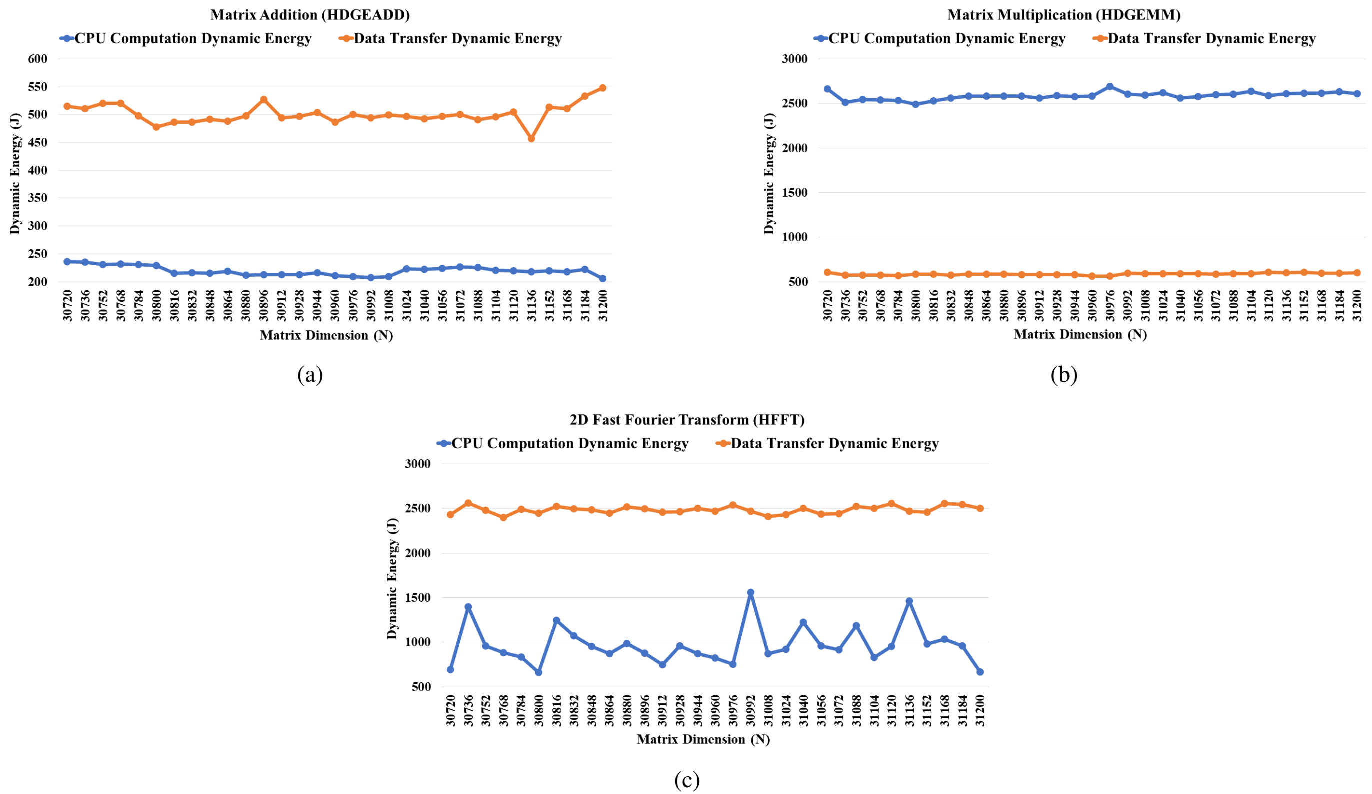


(c) Hybrid parallel 2D fast Fourier Transform application (HFFT) computing the 2D-FFT of a signal matrix S of size $N \times N$. The application comprises three software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). The matrix S is partitioned such that each software component is assigned several contiguous rows of S provided as a parameter to the application. The 2D FFT accomplished in four steps: (1) Computing 1D FFTs on N rows of the signal matrix N. (2) Transpose of the signal matrix. Steps (3) and (4) repeat steps (1) and (2), respectively.

Fig. 13: Description of the three heterogeneous hybrid applications.

Hafiz Adnan Niaz received the B.Sc. degree in computer systems engineering from UCET-IUB and the M.S. degree in computer engineering from the NUST College of Electrical and Mechanical Engineering (CEME-NUST), Islamabad Pakistan. He is currently pursuing the Ph.D. degree from the School of Computer Science, University College Dublin (UCD) Ireland. He is working on optimizing energy of communication for high-performance heterogeneous platforms. His research interests includes parallel computing, high-performance computing, and energy-efficient computing.

Ravi Reddy Manumachu received a B.Tech degree from I.I.T, Madras in 1997 and a PhD degree from the School of Computer Science, University College Dublin (UCD) in 2005. He is currently Assistant Professor in the School of Computer Science at UCD. His main research interests include high performance heterogeneous computing and energy-efficient computing.



(a) (b) (c)

Fig. 14: Dynamic energy profiles showing the comparison between computations and data transfers between CPU and A40 GPUs for three heterogeneous hybrid applications, matrix addition (HDGEADD), matrix multiplication (HDGEMM), and 2D fast Fourier transform (HFFT), executed on the Icelake A40 GPU server.

Alexey Lastovetsky received a Ph.D. degree from the Moscow Aviation Institute in 1986, and a Doctor of Science degree from the Russian Academy of Sciences in 1997. His main research interests include high performance heterogeneous computing and energy-efficient computing. He is currently Associate Professor in the School of Computer Science at University College Dublin (UCD). He is also the founding Director of the Heterogeneous Computing Laboratory in UCD. He authored the monographs Parallel computing on heterogeneous networks (Wiley, 2003) and High performance heterogeneous computing (Wiley, 2009).