Saving Energy without Defying Deadlines on Mobile GPU-based Heterogeneous Systems

Arian Maghazeh Unmesh D. Bordoloi Adrian Horga Petru Eles Zebo Peng
Department of Computer Science, Linköpings Universitet, Sweden
E-mail: {arian.maghazeh, unmesh.bordoloi, adrian.horga, petru.eles, zebo.peng}@liu.se

ABSTRACT

With the advent of low-power programmable compute cores based on GPUs, GPU-equipped heterogeneous platforms are becoming common in a wide spectrum of industries including safety-critical domains like the automotive industry. While the suitability of GPUs for throughput oriented applications is well-accepted, their applicability for real-time applications remains an open issue. Moreover, in mobile/embedded systems, energy-efficient computing is a major concern and yet, there has been no systematic study on the energy savings that GPUs may potentially provide. In this paper, we propose an approach to utilize both the GPU and the CPU in a heterogeneous fashion to meet the deadlines of a real-time application while ensuring that we maximize the energy savings. We note that GPUs are inherently built to maximize the throughput and this poses a major challenge when deadlines must be satisfied. The problem becomes more acute when we consider the fact that GPUs are more energy efficient than CPUs and thus, a naive approach that is based on maximizing GPU utilization might easily lead to infeasible solutions from a deadline perspective.

1. INTRODUCTION

Over the last 24 months, programmable GPUs have penetrated mobile platforms providing embedded software designers with a powerful programmable GPU engine. Vivante's embedded graphics cores, ARM's Mali graphics processors, the StemCell Processor from ZiiLabs (Creative) are some examples of low-power embedded GPUs targeted for mobile and embedded devices. In fact, the arrival of embedded GPUs has given the software industry an opportunity to usher in the era of heterogeneous computation for embedded devices as well [1]. Applications to benefit from this include image processing (e.g., convolution), pattern matching and cryptography [2] that are common in embedded systems like intelligent cars [3], augmented reality in wearable glasses and unmanned aerial vehicles. However, many of these embedded systems involve real-time deadlines where temporal constraints must be satisfied. Several questions — on how to systematically distribute computation between the heterogeneous cores, such as scalar CPU cores and throughput oriented GPUs, so that real-time software can be run in a timely fashion — remain un-answered. As we will show in Section 5, the problem becomes even more challenging if minimizing energy consumption is one of the design goals. The overarching goal of our work is to bridge this gap and this paper is an attempt in this direction.

ESWEEK'14, October 12 - 17 2014, New Delhi, India

http://dx.doi.org/10.1145/2656075.2656097

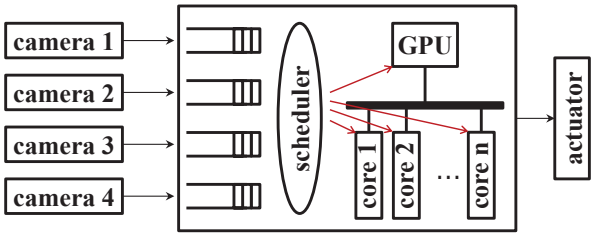

Figure 1: A high-level schematic of a multi-camera based driver assistance system on a heterogeneous platform.

1.1 Overview of our problem

We consider the following set-up that is evident in a wide variety of application scenarios. There are multiple sources of data that periodically stream items of data. The period of each stream depends on the characteristics of the source, among other factors such as the communication infrastructure, and thus, periods of two different streams need not be identical.

Each data item must be processed within a pre-specified deadline relative to the arrival of the data-item. All data items from all streams need to be processed with exactly the same functionality and the relative deadline is the same for all data-items. The set-up described above is very typical for several applications that are targeted towards computation on GPUs. In the following, we highlight two such scenarios.

Scenario 1: Several computationally intensive applications in intelligent cars, and in Advanced Driver Assistance Systems (ADAS), in particular, are amenable for computation on GPUs [4] as well as CPU cores. Some examples of such applications like pedestrian detection, collision-avoidance, blind spot detection and rear lane departure warning are based on a set-up where multiple cameras stream data for sensing the surroundings and environment characterization [5]. The rates at which the camera captures images and streams the frames vary from camera to camera due to a variety of factors like device characteristics of the camera, criticality of the camera position relative to the movement of the car, and so on. Figure 1 shows a high-level overview of such a system.

Each of the frames then goes through a common application such as Image Capture and Transform, Motion Estimation, Object Detection and Classification, among others. It is critical that the system responds to the input frame captured by a camera within a pre-specified deadline. This deadline and periods typically vary for different modes of the system. For example, if it is a brake-by-wire application in an autonomous car, the deadline and periods are different speed slabs. Thus, several such pre-defined modes might be

given and the approach laid out in this paper is targeted towards each of the modes.

Scenario 2: Another scenario arises in the context of secured vehicular communication. The key idea in vehicular networks is to allow vehicles to connect to each other and to a roadside infrastructure to form a vehicular ad-hoc network [6]. The nodes of such an ad-hoc network are commonly divided into two categories. First, there are On-Board Units (OBU), that are radio enabled devices installed on vehicles. Second, we have the Road Side Units (RSU), that constitute the network infrastructure and are placed by the roadside. Such vehicular communication is expected to enable the support of several services like traffic information diffusion, diagnostic services for vehicular health and automatic tolling.

The On-Board Units or OBU, thus, typically need to transmit information from various sub-systems of the car out to the roadside infrastructure or the cloud. Moreover, owing to security reasons, all transmissions from OBU must be encrypted. In fact, OBU is similar to a centralized unit that is connected to all existing bus sub-systems and receives frames (messages) from all of them at different rates. These rates vary due to a wide variety of reasons such as the bandwidth of the underlying bus protocol, speed of processors on the sub-system, characteristics of the host applications and so on. Once OBU receives a frame, it must encrypt it and then transmit it via the radio within a pre-defined time interval relative to the arrival of a frame. In this scenario, note that the same application (i.e., encryption) is processing all message streams coming from multiple sources.

2. RELATED WORK AND OUR CONTRIBUTIONS

Over the last few years, GPUs have started to gain attention in the real-time community [7, 8, 9, 10, 11]. Approaches to specific applications such as medical imaging have also been proposed [12]. Off-the-shelf commercial drivers for GPUs provide no support for real-time scheduling. These papers have taken commendable strides in addressing this issue, both theoretically and in practice. They are on the verge of building robust frameworks that would allow scheduling of GPU memory transfers and GPU computations in a predictable manner. In their perspective, GPU is treated as a shared resource in the context of a multi-tasking environment where some tasks have higher priority than others. A few of the papers introduced the principles of managing the GPU in light of this and others built the required infrastructure based on those principles. The ground-work built by them serves well to complement our proposed approach in the sense that it might be possible to adapt these tools for an embedded GPU that is scheduled according to the scheme proposed in our paper.

Our work has significant differences compared to the above papers. First, we focus on a set-up where the same functionality or task needs to serve streams of data arriving from multiple sources. The key challenge here is to decide which set of data items must be batched together and sent to the GPU for computation. This is very different from the above thread of work where jobs sent to GPUs were assumed to be always of a fixed size. Second, we consider a *mobile* platform where energy efficiency is paramount. None of the papers above addressed concerns related to energy. Third, our system model does not assume a priority-based scheduler and rather, we propose to build a static schedule which is one of the common approaches in the context of periodic streaming applications.

A second set of papers [13, 14], also address the problem of real-time scheduling of parallel data streams on GPUs. However, these works target high performance computing and they completely ignore the energy factor. Also, the underlying assumption on CPU-GPU architecture is different because in their set-up there is a significant overhead in data-transfers between CPU and GPU. In contrast, we target mobile GPUs that share a common memory with the CPU cores where it has been shown that data-transfer overhead is negligible compared to execution time on the GPU [15]. In fact, in future heterogeneous computation architectures, the data transfer overhead between GPU and CPU is expected to be further reduced [1]. This is significant because it implies that a fine-grained scheduling approach, such as the one that we propose, is now feasible without a debilitating interference from data transfer overheads.

There is an emerging line of work [16, 17] that attempts to estimate the wost-case execution time of a task/application running on a GPU. We would like to clarify that we do not focus on computing the worst-case execution time (WCET) which is an orthogonal issue relative to our research. Rather, we assume that the WCET of an application on the GPU is provided to us as an input. As acknowledged in the papers [16, 17] and in their assumptions, it is still difficult to *formally* provide a tight upper bound on the WCET owing to a variety of factors. Rather, they only provide an estimate of the WCET. As our work assumes that WCET is known, we also recommend that the reader bears this in mind. However, if safe guarantees on WCET may be provided in future, the heuristic proposed in our paper guarantee that the resulting schedule will be a safe one from the perspective of meeting deadlines.

Finally, recently it has been shown that GPUs are potentially more energy-efficient compared to CPU cores [18, 19] despite the fact that GPUs have a higher power rating. Very recently, analytical and simulation models to quantify GPU's energy consumption have been developed [20, 21, 22]. This line of work has been limited to high-end GPUs. Recent papers on general purpose computation on mobile GPUs have shown that GPUs do outperform CPU based computations [15] both on performance and energy dimensions. Concluding, to the best of our knowledge, there has been no prior work that proposed an approach to schedule hard real-time jobs from data parallel streams with the objective of minimizing energy on mobile GPU-based heterogeneous platforms.

It should be noted that we do not propose algorithmic or architecture-specific optimizations for the GPU code. As is common in GPU compute programming, we assume that code has already been written or generated [23, 24, 25] and that parameters like size of thread blocks, size of thread groups and other GPU-specific parameters are optimized based on the specific architecture of the GPU.

3. SYSTEM AND APPLICATION MODEL

As discussed in Section 1, we consider a set-up where an application needs to process several real-time data streams and it runs on a mobile GPU-based heterogeneous system.

3.1 System platform

Our heterogeneous platform consists of identical scalar cores, running with same frequencies, on the CPU(s) (henceforth, referred to as *cores* in this paper) and an embedded/mobile GPU. Formally, we say that the platform is composed of a set of processors $P = \bigcup_{i=1,\ldots,m} \{CPUcore_i\} \bigcup GPU$.

3.2 Data stream model

Our system is composed of a set of streams, and each stream generates an infinite sequence of items with a particular period. The set of input streams is denoted by $\mathbf{S} = \bigcup_{i=1,\ldots,|\mathbf{S}|} \{S_i\}$. Each stream S_i has a period h_i and generates a new item I_i^k at each time moment kh_i, where $k = \{0, 1, 2, \ldots\}$. $\mathbf{I}$ is the set of all items $\mathbf{I} = \bigcup_{i,k} \{I_i^k\}$. We assume that this is a synchronous system meaning that all items are initially released at the same time instant when the system starts. The items have to be processed within a deadline D. It is a data parallel system and there is no precedence constraint

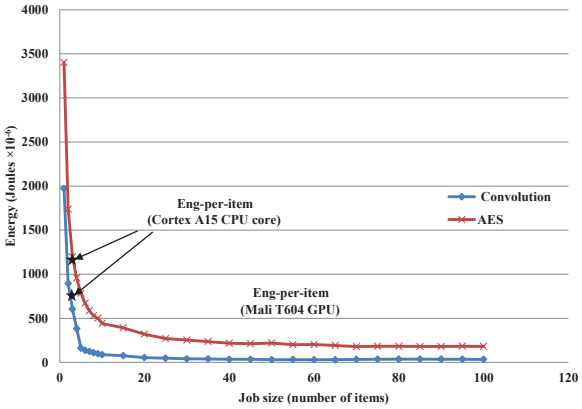

Figure 2: Plots to validate our hypothesis regarding energy consumption per item as the size of the job on GPU increases.

that is imposed between the processing of any two items. Note that this kind of data parallelism is very typical for GPU compute applications.

Each item must be processed either on a core or on the GPU. Each execution on a core or the GPU is called a job. In a core, an item is always processed in a scalar fashion and hence each core job processes a single item. In this paper, we will say that a job on a core is always of size one. On the other hand, items must be packed into a batch of appropriate size and then sent to the GPU that will simultaneously process all the items. If s items are in the batch sent to the GPU, we say that the GPU job is of size s. It follows from our system model that all jobs in the system are identical from functionality point of view but they process different input data. The set of jobs is denoted by $\mathbf{J} = \bigcup_{i=1,\ldots,|\mathbf{J}|} \{J_i\}$.

Items are ready to be processed from the time they are released until they start execution or miss their deadline. Each ready item can be assigned either to one of the cores or the GPU. We assume that jobs are non-preemptive, i.e., once execution starts, it goes to completion until all items in that job have been processed.

T_{core} is the worst case execution time and E_{core} is the energy consumed by a job running on any one of the identical cores. For GPU, however, we have different worst-case execution times and energy consumptions corresponding to different job sizes. T^s_{GPU} denotes the WCET and E^s_{GPU} denotes the energy consumption for a GPU job of size s (including data-transfer for both). We assume that job size s varies from s_{min} to s_{max} and we discuss how the bounds may be obtained in Section 7.1.

4. GPU ENERGY MODEL

As discussed in Section 2, several studies have shown that high-end GPUs, despite being power hungry, are very efficient in terms of throughput per watt. This implies that as the job size increases (until the point where throughput does not increase any further), one may expect better energy efficiency per item. If true, such an insight will help us in designing a schedule for GPU jobs. This needs to be validated, in particular for low-power embedded GPUs.

Recent papers have shown that mobile GPUs outperform CPUs for certain applications [15] both along performance and energy dimensions. However, these studies were limited to an input workload of a fixed-size. It is not known, however, to what extent the GPU will outperform the CPU core as the job size increases. This question is significant in the context of our work where our goal is to schedule launches of different job sizes (that essentially defines the workload) on the GPU. In other words, we ask, what is the impact of job size on a GPU with respect to its energy efficiency?

Towards this, we use energy consumed per item as a metric to quantify the energy efficiency. For CPU cores where items are processed sequentially, job size is always one and this metric has a fixed value. For GPUs, however, we believe that the energy consumption per item varies according to the following hypothesis.

Hypothesis: With an increase of the GPU job size, the energy consumption per item decreases down to a certain level.

The intuition behind this hypothesis comes from the fact that GPUs are throughput-oriented machines. Thus, the execution time on the GPU is not expected to increase linearly with the job size. This phenomenon inherently should lead to better energy efficient per item for larger job sizes on the GPU.

Validation of the hypothesis: We proceeded to test our hypothesis via a set of experiments on a real platform with a mobile GPU. The specification of the mobile GPU is provided in Section 8. Moreover, We selected two GPGPU applications - convolution and Rijndael (from Advanced Encryption Standard) algorithms.

For the convolution kernel, we define an image size of 20x20 pixels as an item while for AES we define an item to be a text file of size 10 KB. We iterated our experiment by varying the job size from 1 item up to 100 items. For each job size, we measured the average current drawn by connecting an ammeter in series to the Arndale board. We followed a standard methodology to measure the currents the details of which are described in [15]. Thereafter, we measured the execution time of the kernel. Having the current, voltage and the execution time, we can easily estimate the energy consumption of a given job size that will directly give us the energy consumption per item.

The results for both AES and convolution are shown in Figure 2. They validate our hypothesis that energy consumption per item decreases as the job size increases. Note that in both cases the gain saturates after the workload surpasses a certain threshold. For convolution, we can observe that after job size increases beyond 25 items, the energy per item does not decrease any further. Similarly, for AES we observe that there is almost no gain in energy per item if we increase the job size beyond 50 items. This phenomenon occurs because after a threshold all computational resources of the GPU are fully utilized and any further increase does not enhance the throughput on the GPU.

The graphs also show the energy consumption for processing one job on a core. In both cases, we can observe that the energy consumed for one item on a core is relatively far worse off than on the GPU even when the energy efficiency offered by the GPU has not reached its saturation point (except when job size on GPU is only 2 items). This implies that, from an energy perspective, it might be worthwhile to send items to the GPU even when we know that the GPU might be under-utilized.

5. THE CHALLENGE

The above discussion on the energy model of mobile GPUs might lead us to conclude that launching as larger jobs as possible, would result in the best schedule in terms of energy consumption. However, this is not trivial at all and in this section, we use three motivating examples to highlight the challenges. For these examples we assume a platform with one GPU and one core.

Example: In the first example, we illustrate that scheduling large jobs on GPU might lead to deadline violation. As shown in Figure 3, we have 3 streams with periods of 1, 2 and 3 time units. Let us consider items released in a time interval equal to the length of the hyperperiod. For simplicity, let us assume that we have one GPU where we may launch jobs of size 8 and size 4 only. GPU jobs of size 8 have WCET $T^8_{GPU} = 7$ while GPU jobs of size 4 have WCET $T^4_{GPU} = 4$. In Figure 3(a), we show a schedule where a GPU job of size 8 is launched. The first item in this job has to wait too long until all the items in the job are ready and hence misses

its deadline ($d = 9$). In Figure 3(b), we show another schedule for the same set of items where only jobs of size 4 are being launched. In this figure, the first items in all jobs meet their deadlines. It follows that all other items will also meet their deadlines (since they arrive later but finish at the same time). The two items not sent to the GPU are sent to the core and it may be verified that they also meet their deadlines. To conclude this example, note that selecting GPU jobs of larger sizes might mean that some items have to wait longer before they are sent to the GPU and thus may violate their deadlines. On the other hand, we know that larger GPU jobs are more energy efficient and this conflicting tradeoff makes our problem challenging.

Example: In the second example, we show that even from the perspective of saving energy, launching larger GPU jobs might not necessarily be the best choice. Let us consider a scenario in which we have to pack 16 items into jobs. We assume that deadlines are such that the following two alternatives always meet the deadlines. In the first alternative, we may launch a GPU job of size 12 and send the remaining 4 items to the core. From our experiments for the convolution kernel described in Section 4, we know that for GPU jobs of size 12, our platform consumed 75×10^{-6} Joules per item while for a job on a core it consumed 720×10^{-6} Joules per item. Thus, the total energy consumption in the first alternative is $(12 \times 75 + 4 \times 720 = 3780) \times 10^{-6}$ Joules.

In the second case, we consider launching two GPU jobs of size 8 each. Again, from our experiments we know that these jobs consume 110×10^{-6} Joules per item. Thus, the total energy consumption in this alternative is $(2 \times 8 \times 110 = 1760) \times 10^{-6}$ Joules. From this example, we observe that having larger GPU jobs might not always be beneficial because the items that are left over for the cores might deteriorate the overall energy consumption.

Example: The third example is similar to the second example described above except that we change the first alternative to consider a GPU job of size 12 and another GPU job of size 4 (instead of sending the 4 items to the core). Again, with our calculations, we observe that the total energy consumed is $(12 \times 75 + 4 \times 380 = 2420) \times 10^{-6}$ Joules. Compared to alternative two, this is still worse. This example shows that choosing a large job and a small job on GPU might be worse than choosing two medium sized jobs given that both alternatives satisfy deadlines. Overall, all these examples highlight the fact that it is not trivial to schedule jobs on mobile GPUs.

6. PROBLEM FORMULATION

Before presenting a solution, we now formulate the problem outlined in the previous sections.

As an input, we are given the system and application model that was presented in Section 3 and, as an output, the goal is to generate a schedule table which guides the static cyclic execution of the system *such that* the total energy consumption is minimized and the deadline D is satisfied for each item. Producing the schedule table implies generating the following:

- The function $\Pi : \mathbf{I} \rightarrow \mathbf{J}$, that, for each item, determines the job it belongs to; $\Pi(I_i^k)$ is the job to which I_i^k has been assigned.
- The function $\Theta : \mathbf{J} \rightarrow P$, that, for each job, determines the processor it is mapped to for execution; for a job $J_i \in \mathbf{J}$, $\Theta(J_i)$ is the processor it is executed on. If $\Theta(J_i) = GPU$, the size of the job, $s_{min} \leq |J_i| \leq s_{max}$. If $\Theta(J_i) \in \bigcup_{i=1,\ldots,m} \{CPUcore_i\}$, then $|J_i| = 1$.
- The start time b_i of each job J_i.

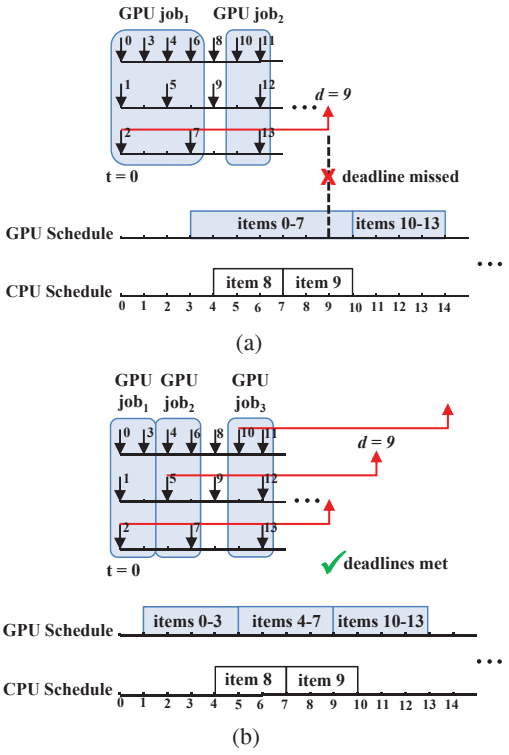

Figure 3: For the same example (a) a GPU job of size of 8 is infeasible but (b) three smaller jobs of size 4 lead to a feasible solution. The arrival of items are shown with downward pointing arrows (in black) while the upward pointing arrows (in red) show the deadlines. In both figures, two items are not part of any GPU job and are sent to cores.

Optimally mapping and scheduling tasks has been shown to be NP-complete even in simple contexts than the one presented above [26, 27, 28]. Therefore, in the following section, we will present a heuristic to address the above optimization problem.

7. HEURISTIC

In this paper, we propose an approach that systematically explores the design space of possible schedules so that a feasible schedule is obtained and the overall energy consumption is minimized at the same time.

7.1 Overview and Preliminaries

As described in Section 3, GPU job size s is bounded between s_{min} to s_{max}. First, we explain how s_{min} and s_{max} may be computed. s_{min} is the smallest job size (i) with lower energy per item than a single core, i.e., $(E_{GPU}^{s_{min}}/s_{min}) < E_{core}$ and (ii) higher throughput than a single core, i.e., $(T_{GPU}^{s_{min}}/s_{min}) < T_{core}$. In our example of the convolution application in Figure 2(b), s_{min} is 3.

Moreover, s_{max} is the largest job size that can be launched on the GPU and still meet all the deadlines of its items. Thus, s_{max} is dependent on the relation between T_{GPU}^{s} and the deadline d. As an example, let us consider two scenarios with deadlines $D_1 = 2ms$ and $D_2 = 3ms$. For convolution, we have $T_{GPU}^{30} = 2.3$, and hence $D_1 < T_{GPU}^{30}$ and $D_2 > T_{GPU}^{30}$. Therefore, s_{max1} must be smaller than 30 while s_{max2} may be larger than 30.

Prior to describing our heuristic, it is important to define the term simulation interval which is the time interval considered by our heuristic to build the static schedule. The method to select the proper simulation interval will be described later in Section 7.4.

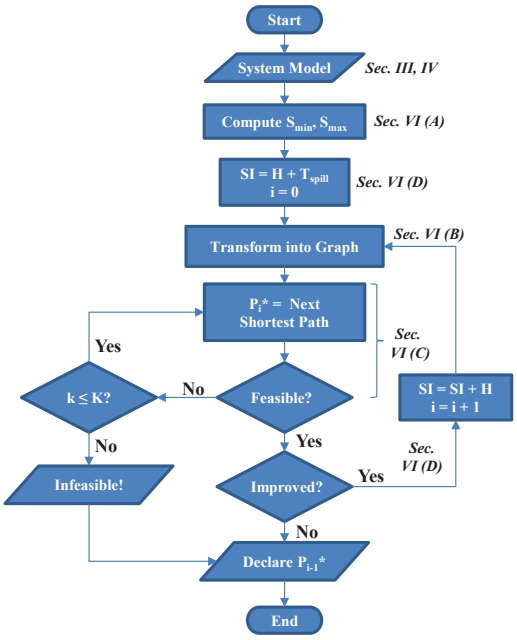

Figure 4: Overview of the proposed heuristic.

The interval starts with the critical instant or the moment where all streams synchronously release their items at $t = 0$. Given a simulation interval and the periods of the streams, the number of items that will be released in this interval by each stream can be trivially computed. For clarity, we assume that each item in this interval is identified by a unique identifier, as shown in Figure 3.

The overview of the heuristic is shown in Figure 4. In the first step, the heuristic intelligently transforms the search space of the static schedules in the given *SI* into a acyclic directed graph. This includes a novel strategy to prune paths that will less likely lead to optimized schedules. Second, the heuristic traverses the graph to find the optimized schedule (shortest path). Thereafter, heuristic iterates over the above two steps by increasing the simulation interval until no more improvements in the solutions can be obtained or a designer specified time-out is reached. As shown in Figure 4, the details for each step are discussed in the following sections.

7.2 Step I: Transforming into a graph

In the first step, we convert the problem of finding the energy-optimized static schedule to a problem of finding the shortest path in a directed acyclic graph. The basic idea is as follows. Any schedule, feasible or not, is essentially a path that consists of some nodes and edges in the graph. A path in the graph represents a particular static schedule which includes the launch times of the jobs on the GPU and the cores. Each edge is weighted based on the energy consumed for the given job(s). Thus, the shortest path in the weighted graph will yield the best schedule in terms of energy consumption. The deadlines also have to be verified, but this is relatively straightforward once the path, i.e., the schedule is known.

Defining the graph: In the following, we describe the significance of our graph in terms of nodes, edges and path and their relation to the original problem.

Nodes: A node n in the graph represents a GPU job that is defined by a tuple (t, l, s). Here, t is the launch time of the GPU job relative to the starting time of the simulation interval ($t = 0$). l is the identifier of the last item in the job and s is the size of the job. In Figure 3(b), the corresponding node for GPU job_1 is $n_a = (1, 3, 4)$.

We also consider two dummy nodes n_s and n_d to be the source and the destination nodes respectively. These nodes do not correspond to any actual GPU launches, rather they are terminal nodes used to cover the total duration of the schedule. For any given problem we have $n_s = (0, 0, 0)$ and $n_d = (t_{end}, id_{last}, 0)$ where t_{end} is the length of the simulation interval and id_{last} is the last item within the simulation interval.

Moreover, the set of items to be dispatched in any GPU job is uniquely identified with l and s. This follows immediately from the fact that given the last item, our heuristic always packs s items into a job in a particular order. This order can be maintained because all items have the same relative deadline and hence, without loss of generality we can enforce an order based on the release times of the items. For our sample node n_a, the job size is 4 and the identifier of the last item is 3 and there is one unique set of items as shown in Figure 3(b). This observation is important because it can dramatically reduce the size of the graph to be constructed.

Edges: An edge exists between two nodes n_i and n_j, if and only if the GPU jobs represented by these nodes may be executed consecutively without any other GPU launch in between. Note that n_j corresponds to a GPU job that consists of items in the range of $[l_j - s_j + 1, l_j]$. Other items in the range of $(l_i, l_j - s_j]$ will be assigned to the first available core immediately after their release. For example, in Figure 3(b), nodes $n_b = (5, 7, 4)$ and $n_c = (9, 13, 4)$ corresponding to GPU job_2 and job_3, are two adjacent nodes. In this example, items 4 to 7 belong to n_b and items 10 to 13 belong to n_c. The remaining items between the two nodes, items 8 and 9, are then assigned to the core immediately after their release time at $t = 4$. Note that the processing of item 9 starts at $t = 7$ after the processing of item 8.

An edge connecting two adjacent nodes n_i and n_j, is further associated a weight representing the total amount of energy required to process the items in the range of $[l_i + 1, l_j]$, that is $E^{s_j}_{GPU} + E_{core} \times (l_j - l_i - s_j)$. In our example of Figure 3(b), the weight of the edge between n_b and n_c is then equal to the energy needed for the corresponding GPU job of n_c and the core jobs, i.e., $E^4_{GPU} + E_{core} \times 2 = 1520 + 1440 = 2960 \times 10^{-6}$ Joules (based on the values obtained from our experiments for the convolution kernel).

Path: Any schedule may then be considered as a path of nodes $p = \{n_s, n_{p1}, n_{p2}, ..., n_d\}$, where n_s and n_d are, respectively, the source and the destination nodes and the rest $n_{pi}, pi \neq s, d$ define GPU launches in the static schedule. Moreover, the order of nodes in a path defines the order of the corresponding GPU launches in the static schedule.

Constructing the graph: In the following, we describe several steps of constructing the graph.

Launch points: The first step of construction is to compute all possible GPU launch points within the simulation interval for all job sizes in the range of s_{min} and s_{max}. For a job size s, the ith launch point lp^s_i is defined by the pair (t_i, l_i) where t and l have exactly the same meaning as they do in a node. In fact, the nodes of the graph are a subset of GPU launch points and they are incrementally added to the graph as it is being built. For example, in Figure 3(b), $lp^4_i = (1, 3)$ is the launch point corresponding to node n_a.

Any launch point, by construction, satisfies the following condition: all items in the corresponding GPU launch will be processed (i) before their deadlines have passed and (ii) before the end of the simulation interval. Formally, for any launch point lp^s_i and any item j in the set of items, $t_i + T^s_{GPU} \leq min(D_j, t_{end})$ where D_j is the absolute deadline of item j and t_{end} is the last time unit in the simulation interval. Therefore, in the example of Figure 3, there is no launch point of size 8.

Furthermore, the same set of items can belong to more than one GPU launch point. For example in Figure 3(b), assuming the length of the simulation interval is larger than $D = 9$, the same set of items in job_1 can belong to any of the launch points $lp^4_1 = (1, 3)$, $lp^4_2 = (2, 3)$, $lp^4_3 = (3, 3)$, $lp^4_4 = (4, 3)$, $lp^4_5 = (5, 3)$.

Adding nodes: In the second step, the heuristic starts building the graph from the source node n_s. Our heuristic always starts with the largest GPU job. One reason for doing this is that in a synchronous system (like ours), the minimum time required to collect any number of items is achieved when the collection starts at $t = 0$. Thus, it is always possible to find a launch point with the largest size (s_{max}) at the beginning of the simulation interval. The other reason is related to a phenomenon known as hyperperiod spill that we will explain in more detail in Section 7.4. Therefore, we first identify the launch points of size s_{max}. Among them, we select ones with the smallest launch time. Note that it is possible to have multiple launch points with similar size (s) and launch time (t) but with different set of items (l). The corresponding nodes of these launch points are then added to the graph to become neighbours of the source node n_s. In Figure 3(b), $n_a = (1, 3, 4)$ is the only neighbour of n_s.

Next, successive nodes are added according to the following policy. Let us assume that node $n_{pre} = (t_{pre}, l_{pre}, s_{pre})$ was the last node created. Then, we know that the end time of the GPU launch corresponding to this node is $t_{now} = t_{pre} + T_{GPU}^{s_{pre}}$. To create next nodes from t_{now}, we have to identify the appropriate launch points that would compose the neighbouring nodes of n_{pre}. Towards this, the heuristic starts from size s_{max} and continues in decreasing order of size until it finds the largest size which has a launch point. The found job size might be smaller than s_{max}. The reason is that as the size increases, the number of launch points decreases. Thus, there might be no launch point of size s_{max} available from t_{now} onwards. Moreover, any job size below the found size will have at least one launch point that may potentially be converted to its corresponding node and later be added to the graph.

However, by adding the nodes of all possible GPU job sizes, the size of the graph might become intractable. Therefore, we propose a tunable parameter ub that may be used by the designer to strike the right balance between the quality of the solution and the running time of the heuristic based on his/her preferences. Essentially, ub is an upper bound on the number of largest GPU job sizes that the newly created nodes can have. Finding the proper value of ub that offers a good trade off between the result and the running time of the heuristic is problem specific.

Next, for each size $s \in \{s_1, s_2, \ldots, s_{ub}\}$, we select only those launch point(s) that have the closest launch time to t_{now}. For each of these launch points lp_i^s, if the number of items which have been accumulated since the previous launch is enough to create a new GPU launch, i.e. $l_i - l_{pre} \geq s$, then a new node $n_{new} = (t_{new}, l_{new}, s_{new})$ is created from this launch point. However, the same node may already be created earlier. Therefore, to prevent increasing the size of the graph with duplicate nodes, a new node is only added when it does not exist in the graph.

We continue with Figure 3(b) as an example. Let us assume that $n_a = (1, 3, 4)$ is the last created node and this gives us $t_{now} = 1 + 4 = 5$. At this point, there are eight launch points of size 4 including $(5, 3)$, $(5, 4)$,..., $(5, 10)$. The first four launch points cannot be converted to a node in the graph because enough number of items has not been accumulated since the previous launch at node n_a, i.e., $(3, 4, 5, 6) - 3 \not\geq 4$. For the next four, however, a new node is created and added to the graph. In the schedule shown in Figure 3(b), node $(5, 7, 4)$ is selected.

Adding edges: A new edge is created between n_{pre} and n_{new}. The weight of this edge is obtained as already discussed with regards to the definition of an edge.

Termination: Moreover, at t_{now} it may happen that no other new node can be created from the available launch points. This means that the current path cannot continue further and n_{pre} should be connected to the terminal node n_d. Finally, the remaining items are considered to be processed on the cores. The corresponding energy of these items are integrated in the weight of the edge which connects the last node to n_d.

To conclude, we consider a graph of a relatively more complex example, the nodes of which are shown in Figure 5. To avoid a crowded figure, only a subset of the edges are shown. The red arrows show a typical path between the source and the destination nodes including the intermediate nodes n_2, n_6, n_{13}, n_9. Each node is placed at the same time unit as its corresponding GPU launch point. The blue and the black lines, respectively, highlight the GPU and the core execution intervals. Consider the nodes n_2 and n_6. At $t = 4$, the job of n_2 with size 12 is launched and by the time it ends, 10 items $(21 - 11)$ have been released. Given that n_6 allows a job of size 8, the remaining 2 items are then left for the cores.

7.3 Step II: Finding the optimized schedule

Once the graph is built, the heuristic finds the path p^* (representing a schedule) with the smallest total weight (representing total energy consumption). Then, it verifies the feasibility of p^* by checking that all the items in the corresponding schedule will (i) meet their deadlines and (ii) be processed before the end of the simulation interval. This is straightforward because the path already contains the information about the launch times of all the jobs. If the schedule is not feasible, the next shortest path is selected.

Note that each path, by definition, encodes Π (the function that maps items to jobs), Θ (the function that maps jobs to processors) and the start time, b, of each job. Thus, the shortest feasible path yields the output desired by our problem formulation (Section 6).

As an example, let us assume that in Figure 5, $D = 14$ and the shown path is a shortest path. None of the items misses its deadline. However, the processing of the last two items on the core needs to be continued beyond the simulation interval which is 38 time units. Therefore, this path does not produce a feasible schedule and the next shortest path must be investigated.

The process of finding the next shortest path continues until the first feasible schedule is found or K shortest paths have been examined. Our experiments validate that typically a small value of K is sufficient. In other words, if no feasible schedule was found, increasing K to very large values will not help. This follows intuitively from the nature of our heuristic. The shortest path, typically, provides the schedule with fewer jobs on the cores. Thus, as we increase K, we find paths where more and more items are sent to the cores and this quickly leads to infeasible solutions.

To find the shortest paths, we use *Yen's K-shortest paths* routing algorithm [29]. Their algorithm starts by finding the shortest path. For this, we use Dijkstra's algorithm [30].

7.4 Step III: Simulation interval

The above process of finding the optimized schedule was completed assuming a fixed simulation interval (SI). A natural choice for SI is the hyperperiod H. However, $SI = H$ might mean that we are ignoring many feasible and/or optimized solutions. This is because we assume arbitrary deadlines which may lead to what is known as hyperperiod spill [31]. Spilling occurs when the items of a hyperperiod are not serviced within their own hyperperiod and are delayed to the subsequent one. Thus, SI would ideally be a time interval such that the schedule is guaranteed to repeat itself after this interval without any spill-over. Below, we provide the background on some work on arbitrary deadlines followed by examples to explain the challenges that arise in our context and finally, our technique for selecting SI in light of these challenges.

Background: Studying the periodicity of the schedules in real-time systems has attracted significant attention in the past. For any feasible schedule generated by a deterministic and memoryless scheduler, the following results are found. In constrained deadline systems on uniform multiple processor systems, if the tasks are synchronous, the schedule will repeat after one hyperperiod [32]. An-

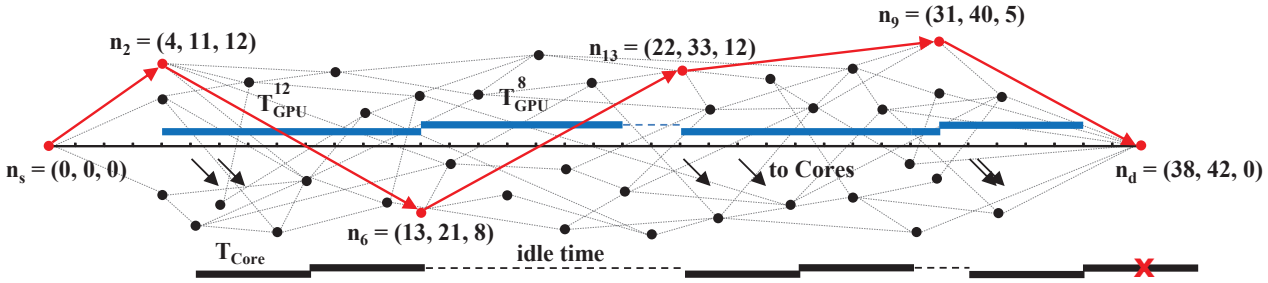

Figure 5: This example illustrates a path in a sample graph. The points are representing the nodes in the graph and the arrows are representing the edges between two consecutive nodes in the path. There are one GPU and one core in this setup and the execution times are $T^{12}_{GPU} = 9$, $T^{8}_{GPU} = 7$, $T^{5}_{GPU} = 5$ and $T_{Core} = 4$.

other study shows that in the case of arbitrary deadlines systems on identical multiple processors, the schedule is finally periodic [33]. In a recent work [34], authors propose an upper bound for such a schedule to reach a cycle. All these results hold for dynamic schedulers like earliest deadline first and fixed priority. In contrast, we focus on building static schedule and hence the above results are not applicable.

Challenges: We use the example in Figure 6 to show the challenges with respect to finding the optimal SI in our setting. Figure 6(a) shows a schedule with $SI = H$, where a job of size 12 is launched on GPU. The GPU is busy serving this job until the end of the hyperperiod. Thus, as seen in Figure 6(a), at the end of the hyperperiod, there are many items left which could not be sent to the GPU. These items may be scheduled to run on the cores but then their execution times spill over SI. However, if we set SI to $2H$, more of these items will be processed by the GPU and less items are left for the cores. This is shown in Figure 6(b). However, even in this case, the execution on both the GPU and the cores go beyond SI. This might potentially be in conflict with the next iteration of the schedule. We propose to solve this problem by introducing a spill-over time (T_{spill}) as explained below.

Solution: The goal is to avoid conflict on resource usage across two cycles. Cycles here refer to repetitive invocations of the static schedule. Note that the spill from one cycle into the subsequent cycle, only causes a problem when there is a conflict of resources between two cycles. In other words, the GPU or the cores may be used by one cycle as long as they are not needed in the next cycle.

Thus, the key idea is that the first use of a resource in a cycle should be delayed as much as possible without jeopardizing schedulability. Towards this, we always launch the GPU with the largest possible size at the start of the schedule. This way, it takes more time to collect items to create the first GPU job in the "spilled-over" cycle. Meanwhile, the GPU job(s) from previous cycle may continue to execute. We denote this overlapping time interval as T_{spill}. As shown in Figure 6(b), $T_{spill} = 5$ time units. During this time, items 38-49 from Cycle 1 are being processed by the GPU and this overlaps with the collection of items 56-67 which belong to Cycle 2. With this, Figure 6(b) now illustrates a feasible schedule. Note that any job invocation on the core will occur only after the first invocation of the GPU. This is because the items collected during T_{spill} are always sent to the GPU. As a consequence, any conflict due to the utilization of the same cores by two consecutive cycles is resolved.

It should also be noted that, increasing SI (to include multiple Hs), does not necessarily lead to a more optimized schedule. For instance, in Figure 6, having $SI = 3H + T_{spill}$ will result in the following schedule. The first two hyperperiods will have a schedule as shown in Figure 6(b) and the third hyperperiod will have a schedule as shown in Figure 6(a). Here it turns out that even the extended SI is not enough to avoid the resource conflict between the two consecutive cycles and, eventually, the execution on the cores spill into the next cycle.

To summarize the solution, our heuristic starts with $SI = H + T_{spill}$. Then it iterates and in each iteration SI is increased by H. This process goes on as long as (i) the heuristic is able to find a feasible schedule for the current iteration and (ii) an improvement (or a minimum threshold for an improvement) is obtained over the previous iteration. Otherwise, the heuristic stops and reports the last found schedule as the final solution. For instance, in the above example, our heuristic stops after two iterations.

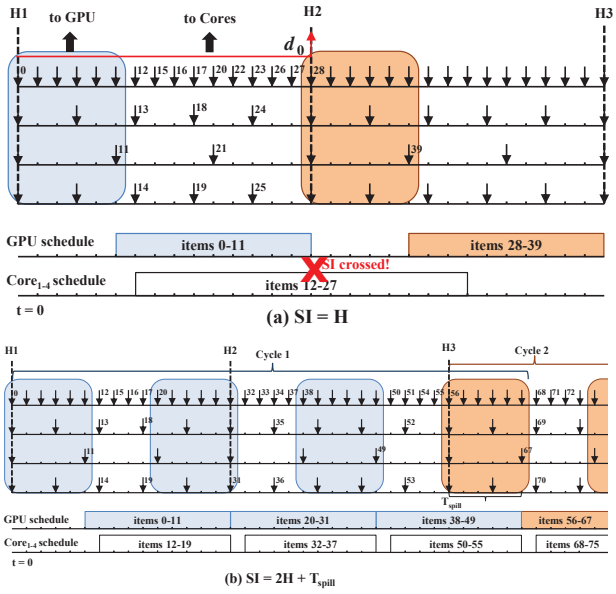

Figure 6: Example to show the importance of the length of SI. Here, there are one GPU and four identical cores. For a system with periods $P = \{1, 3, 5, 3\}$, deadline $D = 15$, $T^{12}_{GPU} = 10$ and $T_{core} = 4$ (a) leads to an infeasible schedule while (b) produces a feasible schedule.

8. EXPERIMENTS

8.1 Experimental setup

We generated 16 problem sets, for which the execution times on the core and on the GPU, as well as the energy consumptions were based on real measurements (Section 4). For this, we selected Samsung Exynos Dual Arndale platform running at 5 volts. The board includes two Cortex A15 cores running at 1.7 GHz and Mali T604 GPU with 4 cores running at 533 MHz. The periods were varied between 0.5 and 8 milliseconds (ms) and the deadline was

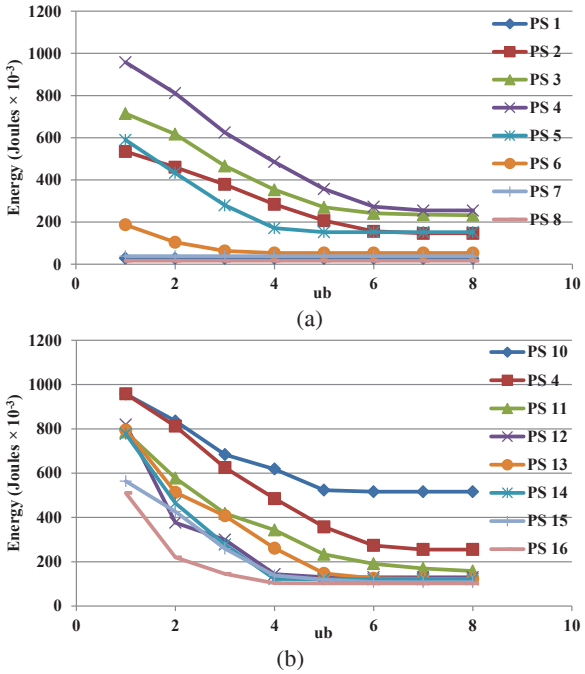

Figure 7: Impact of ub on the energy consumption of the problem sets with (a) same deadline and varying periods, (b) same periods and varying deadline.

varied between 2 and 5.5 ms. The number of streams was set to 10. We set the number of cores to be 4. The upper bound on the number of iterations of the heuristic was set to 5. In the following, we report the results obtained for the convolution application only. We got similar results for the AES application, but omit them due to space limit.

Our heuristic was implemented in C++. All simulations were conducted on a Ubuntu 12.04 64-bit machine with a quad-core Intel Xeon processor running at 2.67 GHz and with 8 GB of RAM.

8.2 Results

In our heuristic, we proposed ub as a knob to obtain higher quality results at the expense of increased running times. We conducted a set of experiments to demonstrate the impact of increasing ub on energy savings. Figure 7(a) shows the results for 8 problem sets where all of them had the same deadline but varying periods. Figure 7(b) shows the results for 8 problem sets where all of them had same set of periods but varying deadlines. Problem set 4 (PS_4) is common in both figures.

All the above problem sets were ran with $k = 10$ and only one (PS_9) was reported to be infeasible from the point of view of deadlines. PS_9 remained infeasible even when k was increased to 100. This is expected as discussed in Section 7.3.

Most of the problem sets in both figures show savings in energy with larger ub except three of them in Figure 7(a). This is expected because for these three, the number of items in one hyperperiod was considerably lower than the others (e.g., 200 compared to around 1100). This implies that finding an optimized solution for these problem sets is relatively easier (design space to be searched is limited) and even with $ub = 1$ the heuristic is able to find a very optimized schedule.

For all 15 problem sets discussed above, we have also plotted the average improvement on energy consumption delivered by $ub = 8$ over $ub = 1, 2, 3$ to 7. As seen in Figure 8, we obtain more than 100% improvement over $ub = 1, 2$ and the improvement almost saturates as we reach $ub = 6$.

Of course, the running times of the heuristic increases with larger

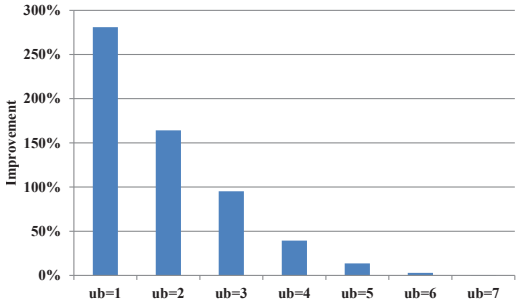

Figure 8: Improvement obtained with $ub = 8$

	ub = 1	ub = 2	ub = 3	ub = 4	ub = 5	ub = 6	ub = 7	ub = 8
PS1	0,39	0,05	0,07	0,16	0,38	1,02	3,12	11
PS2	0,09	0,3	0,94	3,13	8,72	17	27	32
PS3	0,12	0,41	1,78	7	18	53	119	194
PS4	0,2	0,49	2	7,3	37	155	958	3209
PS5	0,09	0,3	1,32	5,28	20	76	432	2236
PS6	0,07	0,18	0,44	0,96	2	6	50	219
PS7	0,1	0,17	0,29	0,42	0,78	1,58	3,14	7
PS8	0,02	0,04	0,04	0,05	0,06	0,08	1,52	1,74
PS10	0,12	0,51	2,85	8,32	29	57,68	124	126
PS11	0,41	1,45	4,48	10,19	35	155	1016	9087
PS12	0,88	3,53	10,15	34	77	252	3602	+10 hs
PS13	1,28	2	4,12	10,73	33	88	180	1354
PS14	2	2,98	8,31	41	90	155	342	4563
PS15	2,83	3,96	6,81	17,85	39	88	155	258
PS16	3,8	5	12,35	33	63	110	194	309

Figure 9: The execution time (in seconds) increases with ub.

ub. This is shown is Figure 9. To make a fair comparison, here we consider only the first iteration of the heuristic by isolating the impact of ub on the running times. For ub values smaller than 6 (which is the saturation point), it can be observed that the heuristic runs in less than five minutes for all problem sets. Moreover, $ub = 8$ led to significantly longer running times (exceeding 10 hours).

8.3 On-board measurements

To further study the performance of the heuristic in terms of energy consumption, we conducted experiments on a real platform (the Arndale board mentioned above). We selected three of the problem sets—PS_4, PS_5 and PS_{14}. For each experiment, we implemented a job dispatcher on the target board. This dispatcher batches the items generated by the streams into jobs and assigns them to the processors based on the pre-computed scheduling table. For each cycle, we measured the overall energy consumption on the board following the methodology in [15].

The key insight exploited by the heuristic is that it leverages CPU-GPU processors in a heterogeneous fashion. To evaluate the energy efficiency of the schedules proposed by our heuristic, we compared the results with a non-GPU version where all items are processed on the cores. The non-GPU version utilizes only one core that sequentially processes the items. The reason to select one core was that the overhead of launching multiple threads on several cores was tremendous and this would give an unfair advantage to our heuristic. It should be noted that the overhead may be compensated if the workload on the core is relatively significant.

Figure 10 reports the average energy consumptions over 10 scheduling cycles, for the proposed schedules and their sequential versions.

	The proposed schedule by the heuristic								Sequential schedule
	ub_1		ub_2		ub_3		ub_4		
	Abs[1]	Imp[2]	Abs	Imp	Abs	Imp	Abs	Imp	Abs
PS4	800	1,3	590	1,7	450	2,3	205	5,0	1020
PS5	420	1,4	250	2,4	150	4,1	127	4,8	610
PS14	824	1,2	447	2,3	221	4,6	70	14,6	1020

[1] Absolute energy consumption (mJ) [2] Improvement factor over sequential schedule

Figure 10: On-board energy measurements

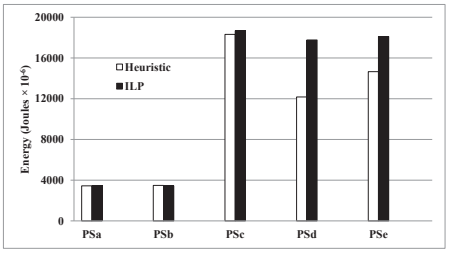

Figure 11: Energy consumption of heuristic vs. ILP (with a time limit of one hour).

It also compares the energy improvements for different values of ub over the sequential schedule. Here, ub_1 and ub_4 indicate the lowest and the highest quality schedules, respectively. As we expected, for each problem set, energy efficiency improves as ub increases. As the best case, in PS_{14} with ub_4—where all the items are processed on the GPU—we get an improvement factor of 14.6. These results show the static schedules generated analytically by our heuristic indeed deliver the expected improvements on real-life applications.

8.4 Comparison with optimal

We also formulated our problem as an ILP (Integer Linear Programming) model and used ILOG CPLEX optimization software to solve the resulting model [35]. In Appendix, we provide an ILP formulation of the problem. We do not discuss several optimizations that we implemented in CPLEX.

We would like to note that the ILP model performs very poorly in terms of scalability. In fact, it was not possible to obtain results from the ILP for the problem sets discussed so far. For most of these problem sets, the number of items in the hyperperiod, which impacts the size of our problem, is in the order of hundreds/thousands. As such, we constructed 5 small-sized problem sets (PS_a to PS_e) that allowed us to obtain some results from ILP for comparison. The number of items in the hyperperiod varied from 29 to 63 for these problems.

Even for such small problem sizes, the ILP did not complete after several hours of running while all results from the heuristic were obtained in less than one second. As such, we set a time limit of one hour on the ILP and report the best results found within this time. Only for PS_a, the ILP was able to find the optimal solution. Note that the heuristic also found the optimum. For PS_b, the best solution reported under one hour was 1% better than the heuristic (see Figure 11). The reason the heuristic delivers slightly worse results is that during the construction of the graph, the first node is chosen greedily to be the largest node (see Section 7). For the rest, the solution found by the ILP was actually worse (see Figure 11). To investigate if we get optimal results without a time budget, we let the ILP run for PS_d. After 12 hours of running, it ran out of memory and the best solution it found until then was the same as the heuristic. These results were obtained with $SI = H$ and, as seen above, it was impossible to run experiments with larger SI. To conclude, the intractability of the ILP highlights the importance of our heuristic that runs efficiently.

9. OUTLOOK AND FUTURE WORK

To the best of our knowledge, this is the first work to address fine-grained scheduling of items from data parallel streams on a mobile GPU-based heterogeneous platform. This work may be extended in several directions.

First, as shown in Section 5, even in the case where the relative deadlines are the same for all items, the resulting optimization problem is non-trivial. Our heuristic was specially customized to the case of similar deadlines. Designing a heuristic for different deadlines remains an open problem. Second, in this work we focused on synthesizing static schedules. Investigating the applicability of other scheduling policies to the same system model might lead to interesting insights. Third, in this work we assumed deadlines can be arbitrarily large and this implied that the length of the static schedule could not be bounded by the hyperperiod. It is worthwhile to find whether or not there exists a theoretical upper bound on the length of the static schedule (SI) that will guarantee schedulability as well as optimal energy efficiency. It will also be worthwhile to extend our work to handle precedence constraints that might exist between items. Finally, in near future we can expect embedded platforms to be equipped with multiple GPUs and it will be interesting to extend our framework to such platforms.

10. REFERENCES

[1] "Heterogeneous System Architecture Foundation," www.http://hsafoundation.com.

[2] Y. Zhang, C. J. Xue, D. S. Wong, N. Mamoulis, and S. M. Yiu, "Acceleration of composite order bilinear pairing on graphics hardware," in *International Conference on Information and Communications Security*, 2012.

[3] M. Lukasiewycz et al., "System architecture and software design for electric vehicles," in *DAC*, 2013.

[4] "OpenCL Computer Vision Products for Advanced Driver Assistance Systems," http://www.vivantecorp.com/media-article.html.

[5] "Freescale Automotive Solutions Brochure: BRAUTOSOL," http://www.freescale.com/ADAS.

[6] P. Cencioni and R. D. Pietro, "A mechanism to enforce privacy in vehicle-to-infrastructure communication," *Computer Communications*, vol. 31, no. 12, 2012.

[7] H. Lee and M. Al Faruque, "GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform," in *DATE*, 2014.

[8] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *RTSS*, 2011.

[9] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "Resource sharing in GPU-accelerated windowing systems," in *RTAS*, 2011.

[10] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *RTSS*, 2013.

[11] G. Elliott and J. Anderson, "Robust real-time multiprocessor interrupt handling motivated by GPUs," in *ECRTS*, 2012.

[12] R. Membarth, J. Lupp, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Dynamic task-scheduling and resource management for GPU accelerators in medical imaging," in *International conference on Architecture of Computing Systems*, 2012.

[13] U. Verner, A. Schuster, and M. Silberstein, "Processing data streams with hard real-time constraints on heterogeneous systems," in *Proceedings of the International Conference on Supercomputing*, 2011.

[14] U. Verner, A. Schuster, M. Silberstein, and A. Mendelson, "Scheduling processing of real-time data streams on heterogeneous multi-GPU systems," in *International Systems and Storage Conference*, 2012.

[15] A. Maghazeh, U. Bordoloi, P. Eles, and Z. Peng, "General purpose computing on low-power embedded gpus: Has it come of age?" in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, 2013.

[16] K. Berezovskyi, K. Bletsas, and B. Andersson, "Makespan computation for GPU threads running on a single streaming multiprocessor," in *ECRTS*, 2012.

[17] A. Betts and A. Donaldson, "Estimating the wcet of GPU-Accelerated applications using hybrid analysis," in *ECRTS*, 2013.

[18] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *International Symposium on Parallel Distributed Processing*, 2009.

[19] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, "Energy-aware high performance computing with graphic processing units," in *International onference on Power aware computing and systems*, 2008.

[20] S. Hong and H. Kim, "An integrated GPU power and performance model," in *ISCA*, 2010.

[21] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator," in *International Symposium on Performance Analysis of Systems and Software*, 2013.

[22] J. Leng, T. Hetherington, A. ElTantawy, S-Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *ISCA*, 2013.

[23] R. Membarth, A. Lokhmotov, and J. Teich, "Generating GPU code from a high-level representation for image processing kernels," in *International Conference on Parallel Processing*, 2011.

[24] H. Jung, Y. Yi, and S. Ha, "Automatic CUDA code synthesis framework for multicore CPU and GPU architectures," in *International Conference on Parallel Processing and Applied Mathematics*, 2011.

[25] S. Ha and H. Oh, "Software synthesis in the ESL methodology for multicore embedded systems," in *SAMOS*, 2011.

[26] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *J. ACM*, vol. 24, no. 2, Apr. 1977.

[27] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *Software Engineering, IEEE Transactions on*, vol. 15, no. 11, Nov 1989.

[28] J. Ullman, "Np-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384 – 393, 1975.

[29] Yen and J. Y, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quart. Appl. Math*, vol. 27, no. 4, pp. 526–530, 1970.

[30] Dijkstra and E. W, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[31] B. Dave, G. Lakshminarayana, and N. Jha, "Cosyn: Hardware-software co-synthesis of heterogeneous distributed embedded systems," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 92–104, 1999.

[32] L. Cucu and J. Goossens, "Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors," in *Conference on Emerging Technologies and Factory Automation*, 2006.

[33] ——, "Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems," in *DATE*, 2007.

[34] E. Grolleau, J. Goossens, and L. Cucu-Grosjean, "On the periodic behavior of real-time schedulers on identical multiprocessor platforms," *Computing Research Repository*, 2013.

[35] "IBM ILOG CPLEX Optimization Studio," http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud.

APPENDIX

A. INTEGER LINEAR FORMULATION

Below we provide the constraints of the ILP formulation of our problem for one iteration. Let $\mathcal{I} = \{1, 2, \ldots, n\}$ be the set of *indices* of items in the simulation interval. Let $\mathcal{J} = \{1, 2, \ldots, n\}$ be the set of indices of jobs—either on a core or on the GPU—and let σ_j, τ_j and ϵ_j be the size, execution time and energy consumption of job $j \in \mathcal{J}$, respectively. Let also $\mathcal{M} = \{1, 2, \ldots, m\}$ be the set of indices of cores.

We define the *job-type* variables $g = (g_1, g_2, \ldots, g_n)$ and $c_m = (c_{1m}, c_{2m}, \ldots, c_{nm})$, $m \in \mathcal{M}$, where $g_j = 1$ and $c_{jm} = 1$, iff jth job is scheduled on the GPU, and on the core m, respectively; otherwise, $g_j = 0$ and $c_{jm} = 0$. We define *item-to-job assignment* variables x_{ij}, $\forall i \in \mathcal{I}$, $\forall j \in \mathcal{J}$, where $x_{ij} = 1$, iff ith item is assigned to jth job and $x_{ij} = 0$, otherwise. Let also b_j denote the start time of jth job. Without any loss of generality, let us assume that all the variables associated with the jobs and items are non–negative integers. Then, the model can be formulated as:

$$\min \sum_{j \in \mathcal{J}} \epsilon_j \tag{1}$$

$$\text{s.t.} \sum_{j \in \mathcal{J}} x_{ij} = 1 \qquad i \in \mathcal{I}; j \in \mathcal{J} \tag{2}$$

$$g_j = 1 \Leftrightarrow s_{min} \leq \sigma_j \leq s_{max}, \qquad j \in \mathcal{J} \tag{3}$$

$$\sum_{m \in \mathcal{M}} c_{jm} = 1 \Leftrightarrow \sigma_j = 1, \qquad j \in \mathcal{J} \tag{4}$$

$$g_j + \sum_{m \in \mathcal{M}} c_{jm} = 0 \Leftrightarrow \sigma_j = 0, \qquad j \in \mathcal{J} \tag{5}$$

$$x_{ij} A_i \leq b_j, \qquad i \in \mathcal{I}; j \in \mathcal{J} \tag{6}$$

$$b_j + \tau_j \leq x_{ij}(A_i + D), \qquad i \in \mathcal{I}; j \in \mathcal{J} \tag{7}$$

$$(g_j + g_k = 2) \Rightarrow b_j + \tau_j \leq b_k, \qquad j, k \in \mathcal{J}, j < k \tag{8}$$

$$(c_{jm} + c_{km} = 2) \Rightarrow b_j + \tau_j \leq b_k, \quad j, k \in \mathcal{J}, j < k; m \in \mathcal{M} \tag{9}$$

$$g_1 = 1,\ c_{(m+1)m} = 1, \qquad m \in \mathcal{M} \tag{10}$$

$$g_j = 1 \Rightarrow b_j + \tau_j \leq b_1 + H, \qquad j \in \mathcal{J} \tag{11}$$

$$c_{jm} = 1 \Rightarrow b_j + \tau_j \leq b_{m+1} + H, \qquad j \in \mathcal{J}; m \in \mathcal{M} \tag{12}$$

$$x_{ij} \leq 1,\ g_j \leq 1,\ c_{jm} \leq 1, b_j \geq 0, \qquad i \in \mathcal{I}; j \in \mathcal{J}; m \in \mathcal{M} \tag{13}$$

where,

$$\sigma_j = \sum_{i \in \mathcal{I}} x_{ij}, \quad \tau_j = \sum_{s=0}^{s_{max}} \begin{cases} T_s & if\ \sigma_j = s \\ 0 & otherwise. \end{cases}, T_s = T_{core} \cup T^s_{GPU},$$

$$\epsilon_j = \sum_{s=0}^{s_{max}} \begin{cases} E_s & if\ \sigma_j = s \\ 0 & otherwise. \end{cases}, E_s = E_{core} \cup E^s_{GPU}$$

The objective function (1) minimizes the total energy consumption of the jobs. Constraint (2) imposes assignment of each item to exactly one job. Constraints (3) and (4) ensure the size ranges of the jobs on the GPU and on the cores, respectively, where s_{min} and s_{max} are the minimum and maximum job sizes on the GPU. Constraint (5) makes sure that the job is empty iff it does not belong to any job-type. Constraints (6) and (7) enforce the timeliness property of the job, where A_i is the arrival time of ith item. Constraints (8) and (9) ensure that there is no execution overlap between two jobs that run on the same resource. Constraints (10), (11), (12) ensure that any job on a resource ends before the resource is required in the next scheduling cycle; we assume that there is at least one job assigned to each resource in every scheduling cycle. Relation (13) imposes the integrality requirements for decision variables.