The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Efficiency and Productivity for Decision Making on Low-Power Heterogeneous CPU+GPU SoCs

Denisa-Andreea Constantinescu · Angeles Navarro · Francisco Corbera · Juan-Antonio Fernández-Madrigal · Rafael Asenjo

Received: date / Accepted: date

Abstract Markov Decision Processes provide a formal framework for a computer to make decisions autonomously and intelligently when the effects of its actions are not deterministic. This formalism has had tremendous success in many disciplines; however, its implementation on platforms with scarce computing capabilities and power, as it happens in robotics or autonomous driving, is still limited. To solve this computationally complex problem efficiently under these constraints, high-performance accelerator hardware and parallelized software come to the rescue. In particular, in this work, we evaluate offline-tuned static and dynamic versus adaptive heterogeneous scheduling strategies for executing Value Iteration —a core procedure in many decision-making methods, such as reinforcement learning and task planning— on a low-power heterogeneous CPU+GPU SoC that only uses 10-15 Watts. Our experimental results show that by using CPU+GPU heterogeneous strategies, the computation time and energy required are considerably reduced. They can be up to 54% (61%) faster and 57% (65%) more energy efficient with respect to multicore -TBB- (or GPU-only -OpenCL-) implementation. Additionally, we also explore the impact of rising the abstraction level of the programming model to ease the programming effort. To that end, we compare the TBB+OpenCL vs. the TBB+oneAPI implementations of our heterogeneous schedulers, observing that oneAPI versions result in up to 5× less programming effort and only incur in 3% to 8% of overhead if the scheduling strategy is selected carefully.

Keywords Decision Making Under Uncertainty · Markov Decision Processes · Value Iteration · Low-power Heterogeneous Computing · Energy Reduction · oneAPI

1 Introduction and Related Work

A Markov Decision Process (MDP) is a standard formal framework for modeling stochastic planning and sequential decision-making under uncertainty in many disciplines, e.g., artificial intelligence, control

Denisa-Andreea Constantinescu
Department of Computer Architecture, Universidad de Málaga
E-mail: dencon@uma.es

Angeles Navarro
Department of Computer Architecture, Universidad de Málaga
E-mail: angeles@ac.uma.es

Francisco Corbera
Department of Computer Architecture, Universidad de Málaga
E-mail: corbera@ac.uma.es

Juan-Antonio Fernández-Madrigal
Department of Systems Engineering and Automation, Universidad de Málaga
E-mail: jafernandez@uma.es

Rafael Asenjo
Department of Computer Architecture, Universidad de Málaga
E-mail: asenjo@ac.uma.es

systems, robotics, logistics, and maintenance, to name just a few [24,4,29,20]. *Solving* an MDP is equivalent to finding a *policy* that determines the best action for each state of a decision-making agent. The policy that maximizes —under certain optimality criteria— the reward for the agent is called *optimal* and can only be obtained by solving the MDP *exactly*. This exactness relies on knowing the true Transition Probability Matrix (T) of the problem, i.e., the stochastic behavior of the decision-making agent while interacting with its environment, a knowledge that can be either given before executing the method or acquired online, during execution [24]. The most used methods for solving MDPs by using explicit knowledge about T, known as *model-based* methods, are based on *Dynamic programming* and include *Value iteration* (VI) and *Policy iteration* (PI). Other sort of methods that may converge asymptotically to the optimal policy (under suitable constraints) in spite of not using explicit knowledge about T include *Temporal difference learning*, *Q-learning* and *SARSA*. They are known as *model-free* methods for learning and decision making, and are frequently applied in Reinforcement Learning [3,20].

In this research, we deal with solving large-scale (with millions of states) tabular model-based MDPs efficiently on low-power computing platforms when using T explicitly. Notice that this can be impractical for large MDP sizes due to the "curse of dimensionality" [2] (i.e., the memory required to represent an MDP increase quadratically with the states, and the time to find the optimal solution, exponentially), while model-free methods spread this computational cost over longer, smaller-grained sequences of experiences of the agent. However, this is not a clear dichotomy [3,10,19], since model-based methods are used in online scenarios as well, and in their approximate forms can employ at their core dynamic programming algorithms —the same as in T-based calculations—, such as VI, for progressively estimating the true T. In particular, in the case of learning in physical environments, e.g., in robotics, model-free methods are recognized not to be as suitable as model-based ones (see Chapter 18 - Reinforcement Learning in Robotics: A Survey in [30]). All of this highlights the importance of dynamic programming, T-based methods, originally devised for offline and exact computation, also in approximate, closer to real-time approaches, and in physical applications.

Although the execution of decision-making algorithms in physical agents (e.g., mobile robotics, autonomous driving) reveals the important problem of having a limited energy supply, most previous works for solving large MDPs use high-performance platforms connected to an uninterruptible power source. In that context, they use one or a combination of the following three approaches to improve efficiency: exploiting parallelism on CPU (SPMD, implemented with OpenMP and vectorization) [23,15, 32,13], exploiting parallelism on GPU (SIMT, implemented with CUDA on discrete GPUs) [23,12,34, 17], and using approximate methods (parallelized for multi-core and GPU execution) [25,15]. Some of the reviewed works consider solving MDPs on low-power platforms [23], but do not evaluate the power dissipation [23,15,32,12]. All these approaches have the common goal to reduce the time required to compute a policy while neglecting the energy footprint. Also, they do not exploit the full potential of simultaneous heterogeneous computing, i.e., they use one kind of device at a time —CPU or GPU, but not both simultaneously.

In contrast, in this paper we target low-power platforms that usually rely on battery power supply; in this scenario, energy consumption awareness is mandatory. Our approach to solving this computationally complex problem is using low-power high-performance accelerator hardware along with multicore processors. The demand for high-performance and energy-efficient computing on consumer mobile devices such as tablets, smartphones, laptops, and gaming consoles has created the perfect conditions for the development of the ubiquitous low-power heterogeneous SoC that integrates a GPU accelerator with a multicore. There is also a growing interest from the scientific community and the industry in low-power Heterogeneous Computing Platforms (HCPs) because they promise improved resource utilization, energy efficiency, and an overall gain in performance, cheaply. Their use in embedded and mobile systems is being extended to solve complex decision problems, e.g., in autonomous driving and service robotics [6, 26,33]. Thus, we illustrate our approach with a common navigation task for a mobile service robot.

Our research group has previously developed techniques that enable the simultaneous execution of the workload of a given software by using both the CPU cores and the integrated GPU on high-performance HCPs to low-power SoCs [7,18]. In this paper, we adapt these previous techniques to investigate whether it is feasible to solve large MDPs in near real-time while minimizing the overall energy consumption, which would be useful for a plethora of decision-making applications for physical agents. Our main goal is to get one step closer to implementing practical and easy to program applications in low-power computing platforms by improving the originally sequential VI algorithm. We illustrate this study with a bare implementation of VI for mobile robot navigation, where a robot is intended to reach some metrical

target from its current position while avoiding obstacles. This is not the best approach for learning this robotic task since we simplify some parts and abstract away several details and components of a complete solution —those not related to the VI procedure itself—, the implemented VI core should be certainly part of more complex methods. Nevertheless, it is a suitable scenario where the benefits of different approaches to reduce the computational and energy costs of decision making can be analyzed and compared.

The implementation of a heterogeneous application that makes the most out of a CPU+GPU SoC is a daunting task due to low-level considerations: data sharing, synchronization, load balancing, scheduling, etc. To make it more approachable, we have developed a high-level heterogeneous parallel_for template [18] that takes care of many of the low-level details, like data partitioning, CPU-GPU synchronization, and scheduling. However, the user is still responsible for providing the implementation of the code that processes a block of iterations (from the parallel iteration space) on the CPU and the GPU. For most of the SoCs with integrated GPU, OpenCL is the main alternative to write the GPU code, and C/C++ the widely accepted language to write the CPU code. This leads to a "dual-source programming paradigm" in which C/C++ has to interact with low-level OpenCL, and the developer has to master two different languages and learn how they interplay.

A "single-source programming paradigm" alternative is the just-released Intel oneAPI [14]: a unified programming model that aims to simplify coding across multiple architectures, including CPUs, GPUs, FPGAs, and AI accelerators. It is an open standard based on cross-architecture language Data Parallel C++ (DPC++). DPC++ incorporates SYCL [11] language and extensions for Unified Shared Memory (USM), ordered queues, reductions, subgroups (on CPU and GPU implementations), and data flow pipes (for FPGAs) support. DPC++ includes an abstraction layer on top of SYCL, which in turn, is a C++ abstraction layer over OpenCL. The main benefit of using oneAPI over OpenCL is the single programming language approach, which enables one to target multiple devices using the same programming model, and therefore to have a cleaner, portable, and more readable code. Clearly, oneAPI eases the development of heterogeneous applications, although it does not automatizes the distribution and scheduling of the workload among the CPU cores and the GPU. To add this capability, in this work, we have re-implemented our parallel_for template on top of oneAPI, and, to the best of our knowledge, this is the first paper experimentally evaluating the pros and cons of oneAPI vs. OpenCL for a heterogeneous application.

Summarizing, the main novelty in this paper is the proposal and performance evaluation of heterogeneous CPU+GPU schedulers based on the recently released oneAPI programming model for writing programs that run on heterogeneous devices, using VI as a case study. Additionally, we compare oneAPI with the canonical framework, OpenCL, both in terms of performance, energy efficiency, and programmability. Our evaluation discusses the impact of the abstraction penalties due to the programming model approach and the scheduling strategy and provides some basic guidelines to help programmers select the appropriate programming model and scheduling strategy for MDP-based solutions suitable for low-power platforms.

The remainder of the paper is structured as follows. In the next section, we explain the MDP formalism and the VI core method that we use as a benchmark, the approaches to be analyzed and studied for improving performance and programmer productivity (Section 2). Then, we present our experimental setup and results of the evaluation of the ease of programming, the computational time, and the energy consumption (Section 3). We end with a discussions and conclusions section (Section 4).

2 The MDP Formalism and the VI Method

In this section, we first provide a condensed formalization of MDPs and VI (subsection 2.1). Next, we briefly describe, as a case study, the application of VI for decision-making in a physical agent: a mobile robot that has to navigate safely (see subsection 2.2). Then we explain the different implementations of VI we have dealt with in this work (subsection 2.3). For a more detailed theoretical treatment of MDPs and related methods, here are several excellent texts on the matter [20, 24, 29].

2.1 Formalization of the Problem

We use here a formalization of a discrete MDP as a tuple (S, A, T, r), where (S, A) is a directed graph, T the so-called *transition*, and r a *reward* function. More concretely, S is a finite set of states where the agent and its environment may be at a given time. A is also a finite set of actions that the agent may take. Each action possibly produces a change of the current state and always gets a certain reward for the decision-making agent. The state transition encodes Markovianity (the next state only depends on the current state and the selected action) and the uncertain dynamics in the system, since it is governed by $T = P(s_{t+1}|s_t, a_t)$. The reward, in turn, is defined by $r : (S \times A) \to \mathbb{R}$, and associates state-action pairs to real numbers. The reward establishes the problem to solve (the goal the agent should attain) in an indirect and quite intuitive form: by optimizing that reward, the decision making policy gets optimal. Rewards can also have uncertainty, but we are particularly interested in the expected reward of being in state s and taking action a, unregarding the next state, which can be defined as $R(s, a) = E_{succ(s)}[r(s, a)] = \sum_{s' \in succ(s)} T(s'|s, a)r(s, a)$.

A *policy* $\pi : S \to A$ provides the action that should be taken at any state. Usually, policies are *stationary*, that is, they always indicate either the same action (deterministic policies) or probability distribution over actions (stochastic policies), given the current state. Here we focus on stationary, deterministic policies. Each such policy π can be assigned a *value* V_π (a vector of real numbers that map to S) that allows us to find the best one with respect to some criteria of optimallity. The most common criterion is the *expected total discounted reward*, that would be accumulated after an infinite number of actions are taken starting at each state s and using π to decide the action at every step: $V_\pi(s) = E_{seq(s)}[r(s_1, \pi(s)) + \gamma r(s_2, \pi(s_1)) + \gamma^2 r(s_2, \pi(s_1)) + \dots]$, being $\gamma \in (0, 1)$ the *discount factor* and $seq(s)$ the set of all possible sequences of states followed from s according to π. The discount factor represents the difference in importance between future rewards and present rewards. The value function has a recursive form if we use the linearity of expectation: $V_\pi(s_k) = R(s_k, \pi(s_k)) + \gamma E_{succ(s_k)}[V_\pi(s_{k+1})]$, or, in more detail: $V_\pi(s_k) = R(s_k, \pi(s_k)) + \gamma \sum_{s' \in succ(s_k)} T(s'|s, \pi(s))V_\pi(s_{k+1})$. This is the so-called *Bellman equation*. In the following we need a more relaxed definition of value function, when the first action is not forced by the policy: $Q_\pi(s_k, a_k) = R(s_k, a_k) + \gamma \sum_{s' \in succ(s_k)} T(s'|s, \pi(s))V_\pi(s_{k+1})$. Actually, $Q_\pi(s_k, \pi(s_k)) = V_\pi(s_k)$, thus $Q_\pi(s_k, a_k) = R(s_k, a_k) + \gamma \sum_{s' \in succ(s_k)} T(s'|s, \pi(s))Q_\pi(s_{k+1}, \pi(s_{k+1}))$, which is recursive again.

Solving a MDP consists in finding a policy that is optimal under some definition of value, such as the expected total discounted reward explained above. In that case it can be demonstrated that at least one policy π^* exists that has a maximum value (it is not required to be unique).

T-based methods for solving MDPs through dynamic programming use the Bellman equation in these forms: $\forall s : V_{\pi^*}(s) = \max_{a \in A} Q_{\pi^*}(s, a)$, $\pi^*(s) = \arg\max_{a \in A} Q_{\pi^*}(s, a)$. These serve to set up iterative optimization procedures. In particular, VI iterates on the value of Q until close-enough convergence to the optimal Q^*, and consequently to V^* and π^*, is achieved. It calculates at each iteration step i a new $Q^i(s, a) = R(s, a) + \gamma \sum_{s' \in succ(s)} T(s'|s, a)V^{i-1}(s')$, then gets V^i and π^i from that Q^i, then repeats everything until, for instance, a suitable proximity between consecutive V occurs.

VI has a superlinear complexity on the number of states and actions, which makes it impractical when that number is large, something common in real applications. Its pseudocode is shown in Fig. 1. It iterates on the execution of two kernels, "Evaluate Policy", with a complexity of $O(|A||S|^2)$), and "Improve Policy" with a complexity of $O(|A||S|)$, until the expected total discounted reward does not improve over a given threshold θ.

The VI procedure of Fig. 1, as in [24], needs as inputs the set S of states, the set A of actions, the transition distribution matrix T specifying *T(s'/s,a)*, the expected reward function R for every state-action pair, $\theta > 0$, a threshold that specifies whether we consider that the algorithm has converged to the optimal policy, and $\gamma \in [0, 1]$, the discount factor. The outputs are the last V and π.

2.2 Mobile Robot Navigation as a Decision Making Problem

We use mobile robot navigation as a case study of the feasibility of our approach. Notice that we are not proposing here a new robot navigation or learning method (there are excellent algorithms that do not involve MDP decision-making or learning at all; for a starting point, see, for instance, [1]), nor a

```
procedure ValueIteration(S, A, T, R, θ, γ)
    assign V^0(S) arbitrarily, i ← 0
    repeat
        i ← i + 1
        for each state s do                                          ▷ Evaluate policy
            for each action a do
                V^i(s) = Σ_s' T(s'|s,a)(R(s, a) + γV^{i−1}(s'))
            end for
        end for
        for each state s do                                          ▷ Improve policy
            π^i(s) = argmax_a Σ_s' T(s'|s,a)(R(s, a) + γV^i(s'))
        end for
    until ∀s|V^i(s) − V^{i−1}(s)| < θ
    return π^i, V^i                                  ▷ Return optimal policy and its value
end procedure
```

Fig. 1: Value Iteration Algorithm: pseudocode for a sequential implementation.

practical solution to solve the robot navigation problem under the decision making perspective. Such a solution would involve more sophisticated methods, possibly based on reinforcement learning (RL), that can be built upon VI, but also add supplementary aspects to consider, which are out of the scope of this work. In our case, for instance, we provide the robot with a pre-built T matrix instead of a progressively estimated one, since we aim to study the VI performance in energy consumption and computation time. RL approaches that make very efficient use of progressive knowledge about the system dynamics can be found, for instance, in asynchronous RL [16].

Our robotic problem has one agent, a non-holonomic wheeled mobile robot called CRUMB [9] with a builtin low power processor. The robot has to navigate through a structured indoor environment realistically simulated to reach a given metrical target (in any orientation) while avoiding obstacles. From a practical view, the target can be the recharging station, a desk where mail has to be delivered, or maybe a moving person. We define an indoor environment as a five by five square meters space surrounded by walls. The inner space may contain any number of obstacles, placed in any position. From an MDP perspective, the goal of the robot is to find an optimal policy to perform such navigation.

The MDP formalism considers *full observability*, i.e., sensor readings determine the state. When it is correctly localized, as we assume here, the sensors of CRUMB can measure its orientation in the universal frame, given by an angle θ, discretized into NT values. It also has a Hokuyo URG-04LX rangefinder, which provides an array of NR distances to obstacles, equally spaced in the frontal area of the robot, each one discretized into NG values. We also assume it can measure the distance d to its target, discretized into ND values. Finally, it calculates the angle a between its orientation and the relative location of the target, discretized into NA values. All in all, the number of states of the resulting MDP is $|S| = NT \cdot NR \cdot NG \cdot ND \cdot NA$, where S represents the state space.

As for the *actions*, CRUMB can take any of the following at any iteration of the sequential decision-making process: stay still (a_0), move forward (a_1), move along a curved trajectory to the left (a_2) or the right (a_3), turn around without displacement (a_4), and move backward (a_5). The number of actions is thus $|A| = 6$, where A represents the action space. All action effects last for a given fixed time, enough to produce all their effects and avoid any non-markovianity caused by the dynamics of the physical system.

T models the robot-environment interaction and has the form of a 3D sparse matrix stored in the CSR (Compressed Sparse Row) format. Thus, it does not store the values for those state transitions that have a zero or close to zero probability to occur.

The robot receives a positive *reward* ($R(s) > 0$) when it reaches the target, and a negative one ($R(s) < 0$) when it collides with obstacles. This is the only information defining the task to accomplish.

We have used the educational version of V-REP (V-REP PRO EDU, version 3.3.2.) [21] Robot Simulator to realistically simulate CRUMB and thus building the T matrix[1]. This software is integrated with the Matlab (R2016b) development environment through a CRUMB toolbox for V-REP [9,27].

[1] A detailed report on the matter is available in [5], Sections 3.2 and 3.3.

2.3 Implementations of VI

In Fig. 2, we show the general structure of our VI implementations. First, the parametric MDP is initialized using the navigation experience from the V-REP simulation stored in *Log file*, the number of actions, *NX*, and the discretization parameters, *NT, NR, NG, ND* and, *NA*. This corresponds to the *MDP Initialization* block of Fig. 2, which encompasses the creation of the T and the rewards matrix, R. Next, given an MDP model, the *Value Iteration Algorithm* executes in a loop until an *Optimal policy* is produced.

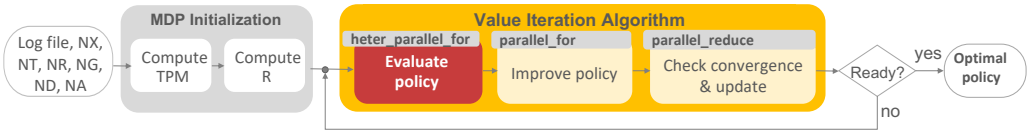

Fig. 2: VI control flow graph.

VI is composed of three kernels with sequential dependencies; therefore, it is necessary to execute them one after the other. We have identified "Evaluate policy" kernel –K1– as the most computationally intensive, followed by the "Improve policy" kernel –K2– (the first two blocks of Fig.2). "Evaluate policy" kernel uses up to 80% of the computing resources, so in this work we focus our attention on optimizing the execution of this kernel. Due to the main data structure that is traversed by this kernel, T, a 3D sparse matrix stored in a CSR format, two challenges arise: i) the memory access pattern is not coalescent; and ii) the computational load of each iteration of the parallel iteration space is different. Therefore, the CPU and GPU threads unavoidably have unbalanced workloads to process regardless of the workload distribution strategy.

2.3.1 Taxonomy of Implementations

In total, we evaluate twelve implementations of VI, incrementally optimized for runtime, energy efficiency, and ease of use. The first version is sequential, SEQ, based on the pseudocode in Fig. 1, and serves as a reference for correctness. The second basic implementation is programmed with OpenCL, OCL, and it is used to run VI only on the GPU. The third variant is optimized for multicore CPU execution; we call it TBB for using the Threading Building Blocks framework (TBB) [28]. It runs the three kernels of VI (Evaluate policy, Improve policy, and Check convergence & update) using *parallel_for* and *parallel_reduce* TBB function templates.

The nine remaining implementations are all heterogeneous, simultaneously exploiting the CPU and the GPU. They are the result of combining three different coding styles for the GPU (OCL, BUFF and USM) and three different heterogeneous schedulers (HO, HD, and HL), that we explain next. Therefore, these implementations are: HO-OCL, HO-BUFF, HO-USM; HD-OCL, HD-BUFF, HD-USM; and HL-OCL, HL-BUFF and HL-USM.

Regarding the three different coding styles, we first have OpenCL, OCL, that represent a low-level programming model in which HW aspects of the GPU are exposed to the user. This implies that the developer has to learn and manage data types as OpenCL platform, device, context, queue, kernel, etc. and deal with low-level functions to compile the GPU kernel, move data to and from the GPU, pass the kernel arguments, enqueue the GPU kernel, etc. These low level details are hidden if we use oneAPI, that offers to coding models: i) BUFF: explicitly declare buffers (that encapsulate the data across the CPU and the GPU) and accessors (that give access to the data and define data dependencies); and ii) USM (Unified Shared Memory): a pointer-based alternative to BUFF, where the data is allocated using `malloc_shared()` and accessed as a regular array from both the CPU and the GPU. This particular data allocation is an Intel extension only available in the Intel's DPC++ compiler (not part of the SYCL standard) and requires HW support for Shared Virtual Memory (a.k.a. SVM or unified virtual address space). The clear benefit is that data movement between the CPU and the GPU is avoided, although cache coherency can be a source of overhead.

Finally, our three scheduler implementations (HO, HD, and HL) have been devised to improve the overall heterogeneous performance. They extend the TBB implementation using increasingly complex scheduling strategies for CPU+GPU heterogeneous computing, and are briefly described next.

2.3.2 CPU+GPU Heterogeneous Computing Implementations

In [18], we propose two scheduling strategies that enable simultaneous execution and efficient use of resources on CPU+GPU platforms, namely Oracle, and LogFit. In [22], we also study a CPU+FPGA scheduler called Dynamic. These three scheduling strategies have been implemented in a library that extends the functionality of the Intel TBB function *parallel_for* to *heterogeneous_parallel_for* by including the possibility to simultaneously orchestrate the work between the CPU cores and the GPU. In these previous works [18, 22], the CPU code is implemented with TBB and the accelerator code with OpenCL. Now we have re-implemented the schedulers on top of oneAPI to ease the development of the GPU kernels. In Sec. 3, we evaluate the ease of using the oneAPI heterogeneous programming model and the overhead introduced by its abstraction layers for both the BUFF and USM models. For completeness, a brief description of the three schedulers follows:

Oracle scheduler (HO): makes a static, one-time, partition of iterations between the CPU and the GPU. HO divides the workload between them using a *RatioGPU* parameter in [0%..100%] to indicate the ratio of the iterations that goes to the GPU. This ratio is set as an input parameter to the scheduler (see lines 9 and 13 in Fig. 3). The scheduler sends the remaining iterations to 100% to the multi-core. To obtain the optimum work balance between the two devices, one should assess every possible partition. For a good approximation of the optimal work division, we have trained the scheduler by executing HO for all work-ratios from 0% to 100% with an increment of 10%.

Dynamic scheduler (HD): works as a dynamic scheduling approach (similar to OpenMP). So the programmer has to set a GPU chunk size, *ChunkGPU*, which is passed as an argument to the scheduler (see lines 10 and 14 in Fig. 3). HD measures the time that the GPU needs to compute a chunk, and estimates another chunk size for the CPU cores that adapts to that time by using a heuristic. This heuristic aims to adaptively set the chunk size for a CPU core by ensuring that it is proportional to the ratio of GPU/CPU-core throughputs (see [22] for details). All devices are dynamically offloaded with their correspondingly sized chunk of iterations until the iterations have been executed.

LogFit (HL): in contrast to HO and HD, which need offline training for optimal partitioning of the workload, we have specially designed HL for irregular applications on heterogeneous CPU+GPU chips. HL has an adaptive partitioning strategy that computes the near-optimal chunk size at runtime, both for CPU and GPU, without user intervention nor previous training. The CPU cores and GPU run at their own pace, while HL adaptively offloads chunks of the remaining iteration to each device so that the overall throughput is maximized. HL computes the CPU chunk in the same way as HD.

For computing the GPU chunk, HL uses a log fitting heuristic (see [18]). This heuristic is composed of an *Exploration Phase* (EP), a *Stable Phase* (SP), and a *Final Phase* (FP). The EP initializes the GPU chunk to the number of Execution Units of the GPU. Next, the EP proceeds in three iterative steps: (1) the GPU chunk size is offloaded to the GPU, (2) the corresponding GPU throughput is measured (and the GPU chunk size recorded —$ChGPU$), and if this throughput improves more than 1% the throughput of the previous GPU chunk, then (3) the GPU chunk is duplicated and going back to step (1). Otherwise, the scheduler transitions to the SP. The SP also proceeds in three iterative steps: (1) it fits a logarithmic curve through the GPU throughput of the previously recorded chunk sizes ($ChGPU$), $a \cdot ln(ChGPU) + b$, and compute its elbow, i.e., the point with maximum curvature. The elbow point of the logarithmic curve allows us to determine a value for $ChGPU$ that is going to be the next optimal GPU chunk size. Then (2) the new GPU chunk size is offloaded to the GPU and (3) the corresponding GPU throughput is measured and recorded. The SP repeats steps from (1) to (3) until the remaining iterations are fewer than the GPU chunk size computed by step (1). In this case, the scheduler transitions to the FP. In FP, if there are sufficient remaining iterations, the scheduler splits them once between the two devices so that they finish the execution at the same time. Otherwise, it sends remaining iterations either to the multi-core or GPU (more details in [18]).

One advantage of the HD and HL is that they adapt better than a static partition as HO in cases of irregular applications although they introduce additional overheads (especially HL) due to the reiterative calls to the scheduler during the partition of the parallel loop and the costs of the fitting operation. Next, we evaluate the performance of these three schedulers for our application in a low-power heterogeneous platform.

2.3.3 Programming Interface

From the programmer perspective, the implementation of the VI algorithm for a CPU+GPU platform implies calling our heterogeneous `parallel_for` template function, that receives three arguments: first iteration; last iteration; and an object of a class that implements the `operatorCPU()` and `operatorGPU()` member functions. These two functions have to implement how a block of iterations are processed on the CPU and on the GPU, respectively.

```
1 int main(int argc, char **argv) {
2   // ViBodyOCL  vib;
3   // ViBodyBUFF vib;
4   ViBodyUSM     vib;
5   // Scheduler params
6   Params p;
7   p.numcpus  = numCPUCores;
8   p.numgpus  = numGPUs;
9   p.ratioGPU = RatioGPU; // Used in HO - Oracle
10  p.chunkGPU = ChunkGPU; // Used in HD - Dynamic
11  ...
12  // Heterogeneous scheduler (hs) HO, HD or HL
13  // Oracle*  hs = Oracle::getInstance(&p);
14  // Dynamic* hs = Dynamic::getInstance(&p);
15  LogFit*     hs = LogFit::getInstance(&p);
16  startTimeAndEnergy();
17  while(notReady) { // Value iteration algorithm
18    // Evaluate policy (Heterogeneous scheduling: CPU+GPU)
19    hs->heterogeneous_parallel_for(0, NS*NX, &vib);
20    // Improve policy, Check convergence & update (TBB-CPU)
21    ...
22  }
23  endTimeAndEnergy();
24  saveResultsForBenchmark();
25 }
```

Fig. 3: Heterogeneous implementations using the Oracle/Dynamic/LogFit Schedulers and OpenCL/oneAPI programming for the heterogeneous kernel.

In Fig. 3, we exemplify the initialization of the three schedulers (lines 5-15), which allows the simultaneous execution of the parallel iterations of the "Evaluate policy" kernel, both on the GPU and the multi-core. The heterogeneous kernel is launched using the `heterogeneous_parallel_for` function, line 19) which receives as input the range of iterations that will be executed ($begin = 0$, and $end = NS \times NX$) and an instance of a functor class implementing the "Evaluate policy" kernel. We have implemented three variants for functor classes: one that uses OpenCL for the heterogeneous kernel code (line 2), and two that use oneAPI for it (lines 3-4) – we give more details on their implementation in Figs. 4 and 5.

We explain the code snippets from Figs. 4 and 5 in parallel, as they are closely related. The first is the functor class used in the *-USM implementations, while the second is its *-BUFF counterpart. Their role is to define how shared memory objects are stored and accessed from the host and the device (see `allocateMemoryObjects` method in Fig. 4: lines 9-11 and in Fig. 5: lines 3-6) and to send work to the GPU and CPU (see `operatorGPU` and `operatorCPU` methods).

In particular, `ViBodyUSM` uses the Unified Shared Memory, or USM, feature of DPC++ to allocate and automatically manage data transfers and synchronization between the GPU and CPU. For our kernel, we need four arrays to represent the MDP model - `probability`, `nextCell`, `nextState`, and `R` - and two more to store intermediary results while computing an optimal policy with Value Iteration - `Q`, and `V`. We represent them all as `X` for brevity (line 10 in `ViBodyUSM` and lines 4, 5, and 9 in `ViBodyBUFF`). All of them are to be accessed for reading from both the CPU and GPU, and `Q` for writing. USM offers three types of allocation: `malloc_device` (can be accessed by the GPU), `malloc_host` (can be accessed by the host CPU and any other device), and `malloc_shared` (like `malloc_host`, additionally, it can migrate to/from the CPU and GPU). The simplest way to meet our requirements is to use `malloc_shared` for all of them (see line 10, Fig. 4). As you see, the USM allocation types have a similar syntax to the standard C/C++ `malloc`. The difference is that they receive one or two extra arguments: the context (`ctx`, defined in line 5, Fig. 4), and additionally, for device and shared type, the device (`dev`, defined in

line 6, Fig. 4). The memory objects allocated like this can be accessed as regular pointers in the kernel code of the CPU and GPU.

`ViBodyBUFF` uses the Buffer abstraction of the SYCL standard for data management. Our six arrays are allocated with `malloc` and later encapsulated in six buffers (Fig. 5, lines 4-5 where again we use `X` to reduce the number of lines in the pseudocode). Buffer's data can be accessed from the CPU or GPU via accessors, which inform the runtime about the access type (e.g., read, write) and about the device that is actually accessing the buffer. For instance, inside `operatorGPU`, in lines 9 and 10, the helper function member `get_access` initialize the GPU accessors: `a_X` (for `X` in {`R`, `V`, `probability`, `nextCell` and `nextState`}) for reading in line 9, and `Q` for writing in line 10. We need to do the same in order to get access for the CPU in `operatorCPU`. We use `sycl::access::mode::discard_write`, abbreviated as `discard_write` (line 10) for `Q`, to point out that the GPU does not need an initialized copy of `Q` because it will be completely rewritten. The other 5 accessors are initialized with the `read` template argument (line 9). There are other access modes available for buffers that we do no use: `write`, `read_write`, `discard_read_write`, and `atomic`.

Once we have configured how data is managed for our kernel, we can send work to the GPU (or CPU). All work requests are done via queues. A queue attaches to a single device (e.g., CPU, GPU, Host, FPGA) and accepts work as a submission (line 13 for `ViBodyUSM` and 8 for `ViBodyBUFF`). We use the SYCL queues `q_cpu` and `q_gpu` to `submit` code to the GPU and CPU for execution. Inside the `submit` call we construct a kernel object (line 14 for `ViBodyUSM` and 11 for `ViBodyBUFF`) passing the data pointers (USM) or accessors (BUFF) to the constructor. With this we can invoke the SYCL `parallel_for` member function of the queue handler `cgh` that will run the code on the device. This

```
1#include "CL/sycl.hpp"
2using namespace cl::sycl;
3queue q_cpu(cpu_selector{});
4queue q_gpu(gpu_selector{});
5auto  ctx = q_gpu.get_context();
6auto  dev = q_gpu.get_device();
7// [ DPC++ with USM ] functor class for heterogeneous_parallel_for
8class ViBodyUSM {
9  void allocateMemoryObjects {                          //6 memory allocations
10   type* X = (type*) malloc_shared(sizeX,dev,ctx); //X = R\Q\V\probability\nextCell\nextState
11 }
12 void operatorGPU(size_t begin, size_t end, event& e){ // send work to GPU
13   e = q_gpu.submit([&](handler& cgh) { // 3 LOC.
14     PolicyEvaluationF kernel{begin,nextCell,probability,V,nextState,R,Q};
15     cgh.parallel_for(range<1>(end-begin), kernel); });
16 }
17 void operatorCPU(size_t begin, size_t end, event& e){} // same as operatorGPU(), uses q_cpu
18 ...
19}
```

Fig. 4: Pseudo-C++ code of the functor class `ViBodyUSM` required by the heterogeneous schedulers to execute the Policy Evaluation kernel of VI. `ViBodyUSM` is implemented with oneAPI & the USM feature of DPC++.

```
1// [ SYCL buffers & accessors ] functor class for heterogeneous_parallel_for
2class ViBodyBUFF {
3  void allocateMemoryObjects {                    // 6 mallocs + 6 buffers
4    type* X = (type*) malloc(sizeX);              // X = R\Q\V\probability\nextCell\nextState
5    buffer<type, 1> buf_X(X, range<1>(sizeX));    // X = R\Q\V\probability\nextCell\nextState
6  }
7  void operatorGPU(size_t begin, size_t end, event& e) { // send work to GPU
8    e = q_gpu.submit([&](handler& cgh) {          // 6 accessors
9      auto a_X = b_X.get_access<read>(cgh);       // X = R\V\probability\nextCell\nextState
10     auto a_Q = b_Q.get_access<discard_write>(cgh);
11     PolicyEvaluationF kernel{a_begin,a_nextCell,a_probability,a_V,a_nextState,a_R,a_Q};
12     cgh.parallel_for(range<1>(end-begin), kernel); });
13   }
14 void operatorCPU(size_t begin, size_t end, event& e){} // same as operatorGPU(), uses q_cpu
15 ...
16}
```

Fig. 5: Pseudo-C++ code of the functor class `ViBodyBUFF` required by the heterogeneous schedulers to execute the Policy Evaluation kernel of VI. It uses SYCL style buffers & accessors.

method can only be called at command-group scope (`cgh` stands for command-group handler). Inside the kernel, data is accessed via pointers for the USM case or via accessors for the BUFF one.

All oneAPI programs must include the "`CL/sycl.hpp`" header (Fig. 4 line 1), and to avoid wordiness, we use the `cl::sycl` namespace for both snippets (Fig. 4 line 2).

3 Experimental Results

Here, we present the performance analysis and a productivity study of our heterogeneous implementations of VI when applied to the case study of the class of decision-making methods that we target in this paper: mobile robot navigation. First, in Subsection 3.1, we describe the experimental setup, including the low-power platform used to carry out the experiments. Next, in Subsection 3.2, we discuss the impact on productivity of the different programming models used in the heterogeneous implementations: OpenCL vs. oneAPI-USM (USM) and oneAPI-BUFF (BUFF). In Subsection 3.3, we analyze the impact on the efficiency of the three programming models, as well as the impact of the scheduling strategies presented previously: static (Oracle, HO) vs. dynamic (Dynamic, HD) and adaptive (LogFit, HL).

3.1 Experimental Setup

We use a *Kaby Lake* platform for implementation and testing purposes. It is a typical heterogeneous platform for mobile environments with low to medium computing capacity, memory, and power-use. Its characteristics are listed in Table 1. We have chosen this platform for two reasons: 1) it is a common customer mobile platform, and 2) it allows us to evaluate the implementability of small to large MDPs.

Table 1: Description of low-power HCP for testing and evaluating our implementations.

Platform	**CPU**	**Integrated GPU**	**RAM**	**TDP (Watts)**	**SW Configuration**
Kaby Lake	i5-8250U @1.60GHz Intel(R) quad core	UHD 620 @300MHz 24 EUs	8GB of DDR4	10 to 15W	Ubuntu 18.04 LTS OS gcc 6.2.0 C/C++ compiler (C++14) OpenCL 1.2, oneAPI 2021.1-beta03 Intel(R) NEO (Gen9) Graphics Driver

We rely on the *Performance Counter Monitor* (PCM) library [31] to monitor performance and energy metrics. PCM gives us access to the hardware counters on Kaby Lake platform, allowing us to measure the *runtime* (in seconds) and the *energy consumption* (in Joules) on the CPU, GPU and Uncore components for a given application. The Uncore, in Intel nomenclature, is the part of the processor containing the integrated memory controller and the Intel QuickPath Interconnect to the other processors and the I/O hub. Time and energy consumption measurements reported in this section have been computed using the average from fifteen executions.

We set a number of MDP benchmarks for autonomous navigation for our experimental study. We create the benchmarks sampling the MDP; in particular, we tackle with the discretization parameters NT, NG, and ND of the MDP, and implicitly, with the navigation precision. The resulting MDPs are exponentially increasing (in exponents of two, approximately) both in the number of states of the MDP and representation size disk: IN8 (2628288 states, 274.7 MB), IN9 (5302368 states, 550.1 MB), IN10 (11520000 states, 1190.1 MB), IN11 (21600000 states, 2229.3 MB), The size of the MDP benchmark that is solvable heterogeneously on a platform is limited by the available RAM and memory of the GPU. For instance, Kaby Lake can handle MDPs that go up to IN11.

In Table 2, we show the mean execution time (in seconds) and energy consumption (in Joules) for the VI execution of the sequential implementation (SEQ), which we use as the baseline for speedup and energy improvement. We show the results from IN8 to IN11 MDP sizes when executing on Kaby Lake.

3.2 Productivity Evaluation: OpenCL vs. OneAPI

In this section, we discuss our findings regarding the evaluation of the Programmability of the programming models used for our implementations: OCL, BUFF, and USM. In particular, we want to characterize how productive from a programmer point of view, are the different approaches. In other words, the ease of

Table 2: Mean execution time and energy consumption of SEQ implementation for benchmarks IN8-IN11. Used as baseline to compare the heterogeneous implementations (Fig. 8).

Execution Time (s)				**Energy Consumption (Joule)**			
IN8	IN9	IN10	IN11	IN8	IN9	IN10	IN11
1.17	2.30	5.07	8.98	14.38	28.43	62.72	111.96

programming of each one. For it, we follow the methodology proposed in [8], where we find, among others, two quantitative metrics to measure the easiness of programming a code: the *Cyclomatic complexity* and the *Programming effort*. The Cyclomatic complexity (CC) is the number of predicates plus one, while the Programming effort (PE) is a function of the number of unique operands, unique operators, total operands, and total operators. The operands correspond to constants and identifiers, while the symbols or combinations of symbols that affect the value of operands constitute the operators. Higher values for both CC and PE metrics mean that it is more complicated for a programmer to code the algorithm.

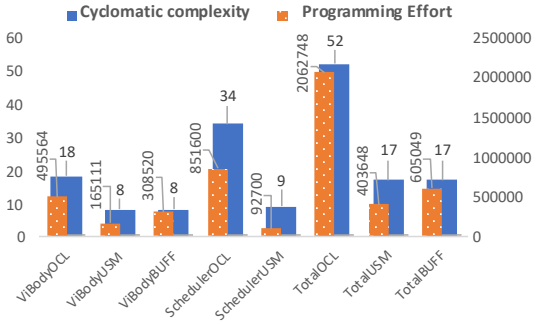

Fig. 6: Evaluation of the Cyclomatic complexity (left Y-axis) and Programming effort (right Y-axis) of OpenCL (OCL) and oneAPI (BUFF, USM) based implementations. The lower, the better.

Fig. 6 shows the results of the Programmability metrics (CC and PE) comparing OpenCL vs. the two oneAPI versions discussed in section 2, USM and BUFF. We break down the metrics for: i) each ViBody functor class (the kernel), ii) the scheduler engine part that is independent of the scheduling algorithm (ScheduleOCL vs. ScheduleUSM[2]); and iii) the total values of each metric when considering the kernel and the scheduler engine. As we see, oneAPI implementations achieve much lower complexity and programming effort than OpenCL. In particular, the most interesting metric, PE, shows that for the kernel implementation, BUFF and USM reduce 1,2x and 3x the programming effort, respectively, when compared to OCL. The reduction is even more significant when considering the whole application: 3,4x and 5,1x, respectively. Interestingly, the programming effort for the implementation of the kernel in USM is 86% lower than for BUFF, although the complexity is the same. Clearly, USM is the implementation with the least effort required.

From a programmer's point of view, the main difference between OCL and oneAPI, when coding ViBody and Scheduler are the following:

ViBody, Scheduler: The kernel source code in OpenCL is defined in a `kernel.cl` file. Next, a program is created using the device context and manually built into a program that can run on the device (`clCreateProgramWithSources`). Finally, we can create a kernel object with another OpenCL API call (`clCreateKernel`), and we are ready to set its arguments and enqueue work to it. When using oneAPI, one only has to submit a functor object to the GPU queue with the code (written in plain C++) that has to execute on the device. Also, the error checking code required for every OpenCL API call (`clCreateBuffer`, `clEnqueueWriteBuffer`, `clEnqueueReadBuffer`, `clSetKernelArgument`, ...) represents a considerable ratio of total lines of code of an OpenCL implementation. In oneAPI, we avoid this by wrapping the code in a try-catch-block, which captures the most frequent error codes and parses them to human readable messages.

Scheduler: All the code needed in OpenCL to get the platform, find the device, create a context and a command queue for the device are replaced by a single line of code in oneAPI (see line 4 Fig. 4).

[2] ScheduleUSM is the same code as ScheduleBUFF

3.3 Efficiency Evaluation: Speedup and Energy Improvement of Heterogeneous Implementations

In this section, we discuss the impact in the performance and energy efficiency, that the different programming approaches have (subsection 3.3.1), as well as the impact of the scheduling strategies proposed (subsection 3.3.2).

3.3.1 Impact of the Programming Model

As we want to focus on the impact the programming model approach has on the performance and energy efficiency, factoring out the effect of the scheduling strategy, in Figs. 7(a) and (b) we compare the time (in seconds) and energy consumption (in Joules) that each programming approach obtains in two scenarios: a) all the workload executes on the GPU, and b) the workload is statically distributed between the CPU multicore and the GPU devices at the beginning of the execution. In this second scenario, the Oracle scheduler (HO) is invoked, and offline training is performed to find the optimal partition between devices. For both GPU-only and HO-* heterogeneous implementations, we find that BUFF implementations are the ones with worse performance, while both OCL and USM present similar good results.

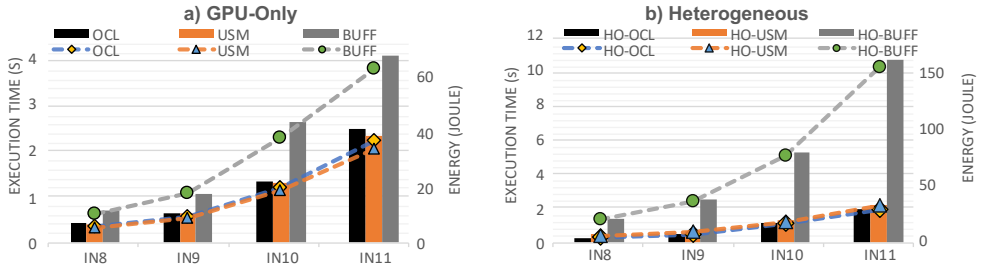

Fig. 7: a) and b) Execution time (left Y-axis, bars) and energy consumption (right Y-axis, lines) for GPU-only and heterogeneous implementations based on Oracle scheduling (HO-*) using OpenCL (OCL), oneAPI & Unified Shared Memory (USM) and oneAPI & Sycl buffers (BUFF). The lower, the better.

In particular, if we focus on GPU-only implementations (Fig. 7(a)) USM exhibits up to 5% and 78% more performance efficiency than OCL and BUFF, respectively, and up to 7% and 97% more energy efficiency than OCL and BUFF. Experiments with heterogeneous implementations based on the Oracle scheduler (HO-*) (Fig. 7.b) reveal that HO-OCL always performs slightly better than HO-USM in terms of energy performance and execution time. They are both up to 380% more energy and time-efficient than HO-BUFF.

As we see, increasing the level of abstraction of the programming model to improve productivity may have a significant impact on the efficiency if we are not careful. In BUFF implementations, increasing the level of abstraction for data management through Buffer functionality helps to hide how different memory locations are mapped on different devices. But it degrades performance because the use of accessors suppose data movements (communication/copy operations). On the contrary, these data transfers are avoided in USM implementations, due to the hardware support of shared virtual memory among devices, so memory locations are accessed directly from the CPU and the GPU.

As USM performs better than BUFF for our set of benchmarks, from now on, we discard the HX-BUFF implementations for further tests and use HX-USM to represent oneAPI.

3.3.2 Impact of the Scheduling Strategy

Now we discuss the impact of scheduling strategies in performance and energy efficiency. Fig. 8 shows the energy improvement and speedup (Y-axis) for the heterogeneous schedulers studied in this work: HO, HD, and HL when solving MDPs of sizes IN8 to IN11 (X-axis) on Kaby Lake. We compute the energy improvement and speedup against the baseline SEQ implementation (see Table 2). For the HO and HD schedulers, we perform offline profiling in which we explore the *RatioGPU* and *ChunkGPU* that achieve the maximum throughput for each input, and for them, we report the speedup and energy improvement we see in the figure. Let us recall that HL adaptively computes the optimal chunk sizes

for the GPU and the CPU cores automatically, without user exploration. Table 3 reports the optimal *RatioGPU* for HO-* schedulers, optimal *ChunkGPU* for HD-* schedulers and average GPU chunk size for HL-* ones. We also show the final GPU ratio for Dynamic (HD-*) and LogFit (HL-*). Interestingly, HL-* schedulers tend to finally offload to the GPU a percentage of the workload similar to the optimal ratio manually found with HO-* (note that this search is done in steps of 10%).

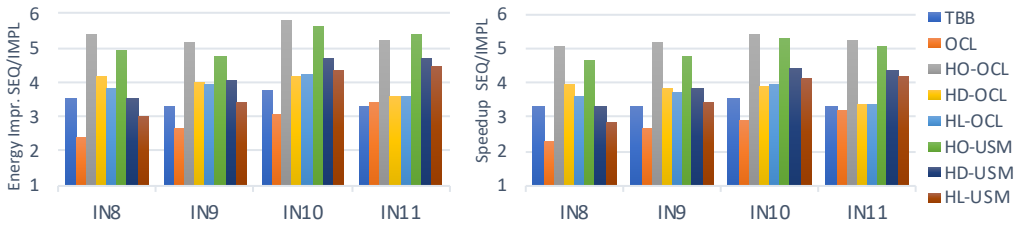

Fig. 8: Energy improvement and Speedup of heterogeneous using Oracle (HO), Dynamic (HD), and LogFit (HL) schedulers implementations with oneAPI (*-USM) and OpenCL (*-OCL) on Kaby Lake. We include for reference the energy improvement and Speedup for CPU-only (TBB) and GPU-only (OCL) implementations. All the results are compared against the sequential (SEQ) code. Higher is better.

As we see from Fig. 8, the time and energy performance results are correlated, i.e., time and energy improve at a similar rate for all implementations and input sizes, although energy efficiency tends to achieve slightly better values than speedup. Also, all heterogeneous implementations outperform CPU-only (TBB) and GPU-only (OCL) implementations for execution time and energy consumption. We see that CPU+GPU heterogeneous strategies can be up to 54% (61%) faster and 57% (65%) more energy efficient when compared to CPU-only (or GPU-only) implementation. We also notice that Oracle scheduling tends to provide the best results, both for OCL and USM programming models. In other words, in spite of the application irregularities, a static partition can provide good results avoiding the partitioning overhead that Dynamic and LogFit incur. For any benchmark size, HO-OCL slightly outperforms HO-USM (see Fig. 9). Both HD-* and HL-* tend to provide similar performance and energy efficiency, although HD-USM and HL-USM obtain better efficiencies than their OCL counterpart for larger benchmark sizes. This is visualized in Fig. 9(a), where we show the execution time ratio of OCL vs. USM for all schedulers and benchmarks sizes from IN8 to IN11. We mark the 1 ratio with a red horizontal line, indicating no difference between the execution time of x-OCL and x-USM. The energy ratios are similar for all inputs (not shown). From this figure, we conclude that for the three schedulers, the overhead introduced by using USM high-level programming over OpenCL decreases with the benchmark size, and as USM achieves better locality with bigger data sets, eventually it outperforms OCL.

Fig. 9(b) assesses the performance degradation of our heterogeneous implementations with respect to the best one: HO-OCL, and for IN8 and IN11. We see that for the larger benchmark size, HD-USM and HL-USM only degrade time by 15% and 18%, and energy consumption by 16% and 19%, respectively, while higher degradation values are seen for HD-OCL and HL-OCL.

An important conclusion that we can draw from this discussion is that for large enough benchmarks, dynamic and adaptive heterogeneous schedulers, benefit more from using oneAPI & USM than from using OpenCL. This probably happens because of the USM feature of oneAPI, which best exploits locality of

Table 3: Optimal workload distribution for different scheduling strategies when executing IN8 to IN11. GPU ratio is shown for HO-OCL and HO-USM; (Chunk size | final GPU ratio) is shown for HD-OCL and HD-USM; (Average GPU Chunk Size | final GPU ratio) is shown for HL-OCL and HL-USM.

	HO-OCL	**HD-OCL**	**HL-OCL**	**HO-USM**	**HD-USM**	**HL-USM**
IN8	70%	4194304 | 53%	3555388 | 60%	50%	1048576 | 31%	1417346 | 62%
IN9	70%	4194304 | 52%	7230334 | 61%	50%	1048576 | 43%	8696996 | 69%
IN10	70%	8388608 | 36%	9907622 | 61%	50%	2097152 | 41%	3800115 | 50%
IN11	70%	8388608 | 37%	11643141 | 60%	50%	4194304 | 45%	3361717 | 52%

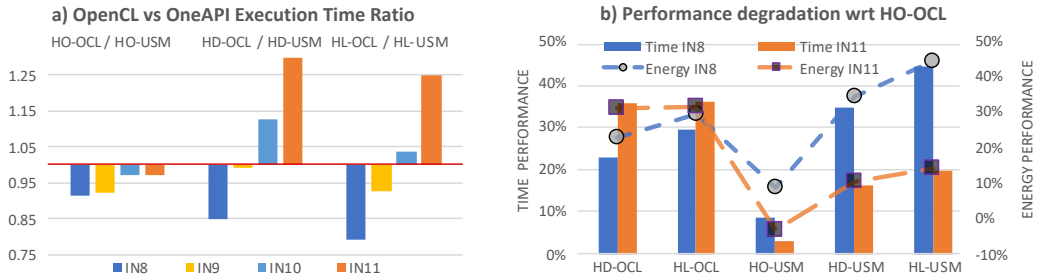

Fig. 9: a) Execution time ratio of OpenCL (x-OCL) vs oneAPI (x-USM) based heterogeneous schedulers (HO, HD, HL). b) Time (bars) and energy (lines) degradation (IN8, IN11) with respect to HO-OCL, the implementation which gives the best energy and execution time results. The lower the better.

shared data thanks to the hardware support of shared virtual memory, while our OCL implementations of both schedulers usually perform some data movement between the memory of the devices.

3.4 Rules of Thumb

Our study allows us to propose a simple set of rules to help programmers select the appropriate programming model and scheduling strategy for MDP-based solutions suitable for using the on-board low-power mobile platform:

- From a productivity's programmer point of view, oneAPI BUFF and USM implementations are easy to program. They result in much simpler codes than OpenCL. However, from a performance and energy efficiency point of view, USM is the choice to go, if the platform supports shared virtual memory by hardware.
- When considering the scheduling strategy, fine-tuned static scheduling (HO) performs best, both from time and energy efficiency point of view.
- From a practical point of view, both static (HO) and dynamic (HD) scheduling require time-consuming offline training to find the optimal GPU ratio/chunk size, which needs to be repeated every time the execution conditions or the input change.
- Adaptive scheduling (HL) performs acceptably with no previous training on any platform, in particular, when based on USM and for large problem sizes.

4 Conclusions

In this paper, we have tested and analyzed the feasibility of solving large-scale Markov Decision Processes exactly with Value Iteration in low-power heterogeneous computing platforms. We seek to widen the applicability of decision-making methods that embed VI as an inner component to real-world problems, so we study the impact on the performance and energy efficiency of implementing VI with different heterogeneous programming approaches and scheduling strategies.

From the three programming models evaluated, OpenCL –OCL–, oneAPI & SYCL style buffers –BUFF– and oneAPI & USM feature –USM–, we learned that oneAPI implementations can be up to 5x easier to program than OCL. Although, increasing the level of abstraction of the programming model for improving the productivity, may have a significant impact on the efficiency if we are not careful: BUFF can be 380% less time-energy efficient than OCL. USM appears as a good alternative if the platform supports shared virtual memory by hardware.

From the three scheduling strategies evaluated, static –HO–, dynamic –HD–, and adaptive –HL– the static scheduling performs best in terms of performance and energy efficiency, though it requires exhaustive offline searching. Adaptive scheduling provides good results with no previous training, in particular when using the USM approach to code the kernels and scheduler, and for large problem sizes.

As future work, we are working to extend our research line by looking into more complex decision-making procedures, i.e., Partially Observable Markov Decision Processes.

Acknowledgements

This work is a result of the research project TIN2016-80920-R, funded by the Spanish Government. It has also been supported by Junta de Andalucía under research projects UMA18-FEDERJA-108, UMA18-FEDERJA-113 and TEP-2279.

References

1. Barber, R., Crespo, J., Gomez, C., Hernamdez, A., Galli, M.: Mobile Robot Navigation in Indoor Environments: Geometric, Topological, and Semantic Navigation, chap. 5, pp. 393–640. Intech Open (2019)
2. Bellman, R.: The theory of dynamic programming. Bulletin of the American Math. Society **60**(6), 503–515 (1954)
3. Bertsekas, D.P.: Dynamic programming and optimal control, vol. 2, 3 edn. Athena Scientific (2007)
4. Boucherie, R.J., van Dijk, N.M.: Markov decision processes in practice (2017)
5. Constantinescu, D.A.: Optimization of a decision making algorithm under uncertainty for heterogeneous platforms. Master's thesis, Universidad de Málaga (2017). DOI 10.13140/RG.2.2.24922.70082
6. Coradeschi, S., et al.: GiraffPlus: a system for monitoring activities and physiological parameters and promoting social interaction for elderly. In: Human-Computer Sysx Interaction: Backgrounds and Applications 3. Springer (2014)
7. Corbera, F., Rodríguez, A., Asenjo, R., Navarro, A., Vilches, A., Garzarán, M.J.: Reducing overheads of dynamic scheduling on heterogeneous chips. arXiv preprint arXiv:1501.03336 (2015)
8. Dios, A.J., Asenjo, R., Navarro, A.G., Corbera, F., Zapata, E.L.: High-level template for the task-based parallel wavefront pattern. In: 18th Intl. Conf. on High Performance Computing (2011)
9. Fernández-Madrigal, J.A., Cruz-Martin, A.M., Aguilar-Moreno, M., Vega, I.F.: CRUMB: Cognitive-robotics-supporting mobile base (Consulted 1st of August, 2019). URL babel.isa.uma.es/crumb
10. Gordon, G.J.: Approximate solutions to markov decision processes. Ph.D. thesis, Carnegie Mellon University Pittsburgh, PA (1999). URL http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-143.pdf
11. Group, K.: SYCL Specification: SYCL integrates OpenCL devices with modern C++, v1.2.1 (2019)
12. Hernandez, B., Pérez, H., Rudomin, I., Ruiz, S., de Gyves, O., Toledo, L.: Simulating and visualizing real-time crowds on GPU clusters. Computación y Sistemas **18**(4), 651–664 (2014)
13. Iannucci, S., Chen, Q., Abdelwahed, S.: High-performance intrusion response planning on many-core architectures. In: International Conference on Computer Communication and Networks (ICCCN), pp. 1–6. IEEE (2016)
14. Intel: Intel oneAPI Programming Guide (Beta) (2019)
15. Jaskowski, W.: Mastering 2048 with delayed temporal coherence learning, multi-stage weight promotion, redundant encoding and carousel shaping. IEEE Transactions on Computational Intelligence and AI in Games (2017)
16. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International conference on machine learning, pp. 1928–1937 (2016)
17. Munir, A., Gordon-Ross, A., Ranka, S.: Modeling and optimization of parallel and distributed embedded systems. John Wiley & Sons (2015)
18. Navarro, A., Corbera, F., Rodriguez, A., Vilches, A., Asenjo, R.: Heterogeneous parallel_for template for CPU–GPU chips. International Journal of Parallel Programming **47**(2), 213–233 (2019)
19. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, second edn. Wiley (2011)
20. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming (Wiley Series in Probability and Statistics). John Wiley & Sons (2005)
21. Robotics, C.: V-REP: virtual robot experimentation platform (Consulted 1st of August, 2019). URL www.coppeliarobotics.com
22. Rodríguez, A., Navarro, A., Asenjo, R., Corbera, F., Gran, R., Suárez, D., Nunez-Yanez, J.: Parallel multiprocessing and scheduling on the heterogeneous xeon+fpga platform. The Journal of Supercomputing (2019)
23. Ruiz, S., Hernández, B.: A parallel solver for Markov decision process in crowd simulations. In: Artificial Intelligence (MICAI), 2015 Fourteenth Mexican International Conference on, pp. 107–116. IEEE (2015)
24. Sigaud, O., Buffet, O.: Markov decision processes in artificial intelligence. John Wiley & Sons (2013)
25. Tai, L., Liu, M.: Mobile robots exploration through CNN-based reinforcement learning. Robotics and biomimetics **3**(1), 24 (2016)
26. Thakur, A., Svec, P., Gupta, S.K.: GPU based generation of state transition models using simulations for unmanned surface vehicle trajectory planning. Robotics and Autonomous Systems **60**(12), 1457–1471 (2012)
27. Vega, I.F.: Development of a programming environment for a simulated TurtleBot-2 robot with a WindowsX manipulator arm through the connection of V-REP and MATLAB. B.Sc. Thesis, University of Málaga (2016)
28. Voss, M., Asenjo, R., Reinders, J.: Pro TBB: C++ Parallel Programming with Threading Building Blocks. Apress (2019)
29. White, D.: Markov decision processes. John Wiley (1993)
30. Wiering, M., Otterlo, M.v. (eds.): Reinforcement Learning. State-of-the-Art. Springer Verlag (2012)
31. Willhalm, T., Dementiev, R., Fay, P.: Performance Counter Monitor (PCM) (Consulted 21st of January, 2020). URL https://github.com/opcm/pcm
32. Wu, Z.: Parallelizing model checking algorithms using multi-core and many-core architectures. Ph.D. thesis, Nanyang Technological University, Singapore (2017)
33. Yamaguchi, U., Saito, F., Ikeda, K., Yamamoto, T.: HSR, human support robot as research and development platform. In: Intl. Conf. on Advanced Mechatronics: toward evolutionary fusion of IT and mechatronics, pp. 39–40 (2015)
34. Zhou, H., Khatri, S.P., Hu, J., Liu, F., Sze, C.: Fast and highly scalable bayesian MDP on a GPU platform. In: Intl. Conf. on Bioinformatics, Computational Biology, and Health Informatics, pp. 158–167 (2017)