WATERS Industrial Challenge 2019

Arne Hamann, Dakshina Dasari, and Falk Wurst, Corporate Research Robert Bosch GmbH

Ignacio Sañudo, Nicola Capodieci, Paolo Burgio and Marko Bertogna, University of Modena and Reggio Emilia

1. Introduction

Automotive E/E architectures are currently undergoing a radical shift in the way they are designed, implemented and deployed. Especially, the computational power and communication bandwidth required for new functionalities, such as automated driving or connected vehicle functions (e.g. path planning, object recognition, predictive cruise control), exceed the capabilities of current compute nodes (mainly micro-controller SoCs) and is leading to a reorganization of automotive systems following the paradigm of so-called centralized E/E architectures that are based on a new class of computing nodes featuring more powerful micro-processors and accelerators such as GPUs.
One consequence of these centralized E/E architectures is that heterogeneous applications will be co-existing on the same HW platform, heterogeneous not only in their model of computation (ranging from classical periodic control over event-based planning to stream-based perception applications) but also in their criticality, in terms of real-time and safety requirements. Ultimately, the burden of integration is shifted from the network to the ECU level and in this regard typically from the vehicle manufacturer to the supplier of the control unit.

In order to cope with design and integration challenge, expressive performance models capturing the heterogeneity of the hardware-software system are needed. The WATERS industrial challenge addresses one example in this context, where the processing power offered by GPUs and their capability to execute parallel workloads is exploited to execute and accelerate applications related to advanced driver assistance systems.

Thereby, we start with relatively simple models that are from our point-of-view, on the one hand, sufficient to derive sensible performance predictions for the presented use-case, and that are, on the other hand, not too far away from established models used in the real-time community, or in other words, there is a chance for existing models to be extended for addressing this challenge.

This document contains the following information:

- A primer on NVIDIA TX2 platform with focus on CPU – GPU interactions, scheduling, memory model, and offloading mechanisms as basis for performance modeling in Amalthea
- A brief description of the challenge questions
- A description of the considered application (a concrete Amalthea performance model will be provide to a later point in time)
- A description of utilized Amalthea performance modeling approaches suitable to description the software and hardware parts including their interaction for the given challenge.

This description is a very good start for groups that want to work on the WATERS Industrial Challenge 2019. However, we expect that many additional questions will arise in the next months. Please use the WATERS community forum to ask those additional questions. We will monitor the forum and answer your questions in a timely fashion.

2. NVIDIA TX2 Platform

2.1. Heterogeneous architecture overview

At a high-level abstraction, embedded heterogeneous SoCs (System of Chip) featuring GP-GPU accelerators are characterized by the following hardware components: 1) CPU 2) Accelerator and 3) Memory hierarchy as seen Figure 1.

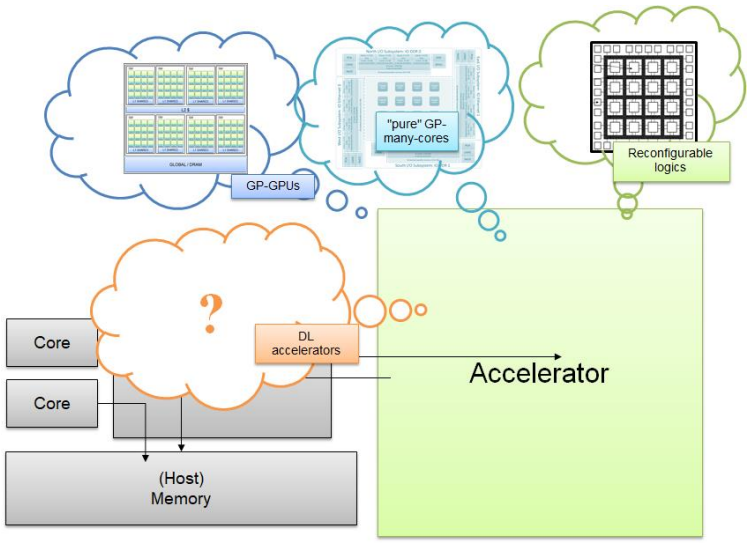


Figure 1: Multi-heterogeneous hardware platform

Host subsystem. As shown in Figure 2, the Jetson TX2 is composed of two different CPU islands, a quad-core 1.9GHz ARMv8 A57 and a dual-core 2GHz ARMv8 Denver (NVIDIA ARM-based proprietary technology). Each of the cores in the A57 subsystem integrates a private 32KB L1 data cache and a 48KB L1 instruction cache, while the Denver feature per-core 32KB L1 data cache and 48KB L1 instruction cache. Moreover, each of the islands has a 2MB L2 cache.

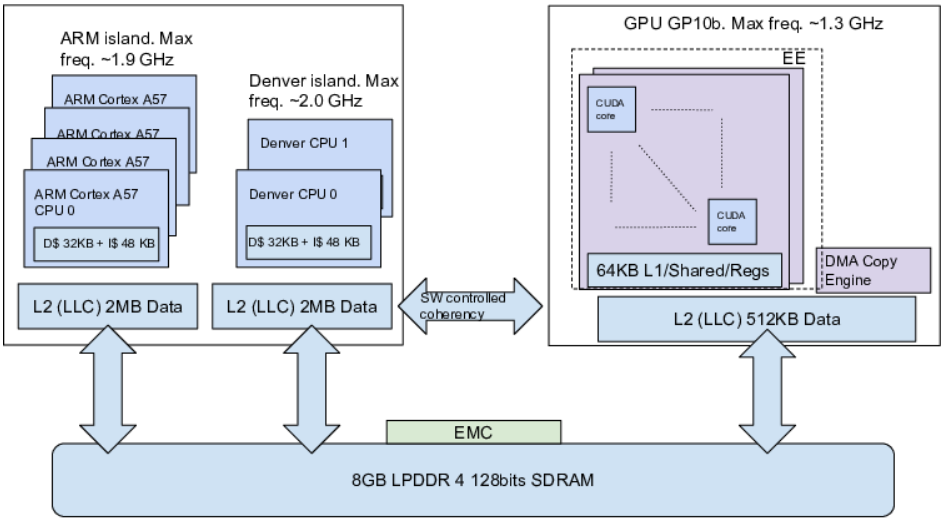


Figure 2: Jetson TX2 hardware architecture

Accelerator. The platform integrates a iGPU (integrated GPU[1]) Pascal-based architecture "gp10b" with 256 CUDA cores grouped within two streaming multiprocessor (SM), each of the SM in the SoC have a 64KB L1 cache, and both SMs share a 512KB L2 cache. The GPU integrates two major hardware

[1] since the TX2 has only one iGPU from now on we will call it GPU

components that are the responsible of the different GPU functions, namely an Execution Engine (EE) that is responsible for performing the parallel workload execution and a Copy Engine (CE) that is responsible of high bandwidth memory transfers.

Memory. In embedded devices, for example NVIDIA Tegra based boards, the GPU and the CPU have access to a common shared memory. Allocations in global memory are managed by the CPU host. The GPU is organized as an array of Streaming Multiprocessors (SM) where the SMs share a common L2 cache. *GP10b has 2 SMs*. With regard to memory, there are two points to consider:

- **Memory bandwidth.** The Tegra X2 SoC features 8GB LPDDR4 128 bit DRAM with a bandwidth of ~58 GB/s. The Denver cores have approximately three times the memory bandwidth of the ARM-A57 cores, i.e. approximately 14 vs 4.5 GB/s, hence memory accesses from the Denver cores have shorter latencies for a single read operation throughout L2 and system RAM accesses. The GPU can access the shared RAM banks[2] with a bandwidth of 20GB/s.
- **Memory allocations.** Memory can be allocated in a pageable & pinned (or page-locked memory) fashion. Accordingly, different programming strategies can be used when transferring data from host to device and vice-versa. In this regard, NVIDIA encourages the use of **pinned** memory (as opposed to **pageable**) to improve the bandwidth of data transfers[3] and support predictability.

2.2. Kernels, Custom Streams and Default Streams

A GPU function, also called command or kernel, refers to a C++ function that is executed on the GPU. An application may be realized with one or more GPU kernels in addition to regular CPU functions. A stream is a queue of both compute and copy operations issued by an application. An application can have multiple streams and typically kernels that can run in parallel are assigned to different streams. Kernels within a stream are executed in FIFO order whereas kernels from two different (custom) streams of the same application can execute concurrently subject to the availability of enough resources on the GPU. Potentially, a kernel on a stream can utilize all the CUDA cores of all the SMs of the GPU. CUDA allows to designate a stream to have either a high priority or a lower priority.

Streams can be categorized as DEFAULT streams or custom streams. When unspecified, all kernels are executed by default in the DEFAULT stream. The default stream is different from other streams because it is a synchronizing stream with respect to kernels on the device: no kernels in the default stream will begin until all previously issued kernels in any stream on the device have completed, and a kernel in the default stream must complete before any other kernel (in any stream on the device) will begin. Custom streams on the other hand can exploit parallelism, can exploit priorities and can clearly specify synchronous or asynchronous execution semantics. *We only assume custom streams in our challenge model.*

2.3. GPU Scheduling Model

In our model, a GPU-based application is a process that creates a so-called *GPU-context*[4]. For instance, two GPU applications identify 2 GPU contexts, which live in different memory spaces, following a standard UNIX-like process execution scheme.

Each application is associated with one (or more) channels. Only a channel associated with one application can execute on the GPU at any given time. So, two applications cannot co-execute at the same time, they may however have interleaved executions. The channels are enqueued in a run-list as seen in Figure 3. The GPU scheduler (implemented in hardware), also called the GPU Host, iterates

[2] http://algo.ing.unimo.it/people/marko/papers/ETFA17.pdf
[3] https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/
[4] GPU contexts for graphic workloads are typically written using OpenGL framework, while general purpose workloads are in C++/CUDA. We focus on the latter.

over the run-list selecting one channel at a time in a weighted round-robin manner. The time for which a channel executes is configurable; if the kernels of a channel complete before its allocated time-slice, the channel is switched out and a new channel is scheduled (work conserving). The number of times that a channel appears on the run-list and whether it can be preempted is also configurable.

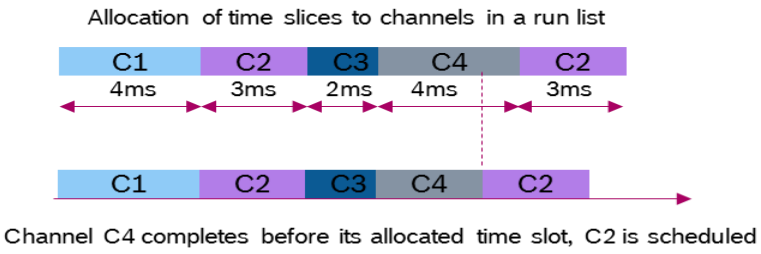


Figure 3: Channel scheduling in a run-list

The timeslice within a channel could be used to execute the copy or the execution phases. When the channel has multiple streams, the copy and execute phases may run in parallel as seen in Figure 4, depending on availability of resources. Please note that in our challenge model, we assume only one stream per application.

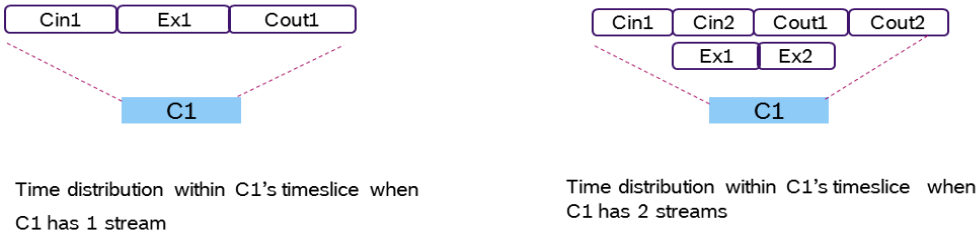


Figure 4: Parallelism with multiple streams

Once a channel is selected, the GPU scheduler dispatches kernels of the selected channel for execution to the SMs (or copy phases to the copy engine). Then, within each SM, its hardware warp scheduler further assigns these commands to execute on different warps (A warp is a group of 32 physical threads, all of which execute in SIMD mode).

2.4. GPU-CPU interaction

A program running on a GPU works in the following way; when a program starts, the CPU copies the data and instructions from host memory domain to GPU memory, this copy is performed using the Copy Engine (CE) as seen in Figure 5. After this operation, the CPU launches one or more kernels: this execution can be either *Synchronous* or *Asynchronous*. If synchronous, the CPU dispatches the work to the GPU and then the CPU waits until the end of the kernel execution. In the asynchronous case, it is possible to perform computations on the CPU, immediately after dispatching the work to the GPU without waiting for completion, in a work-conserving manner.

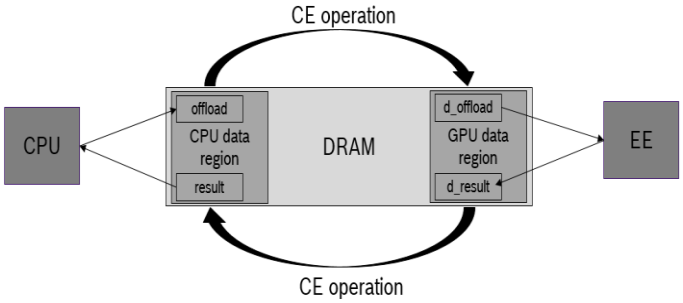


Figure 5: Data transfer handled by the copy engine

2.5. Assumptions

For the WATERS Industrial Challenge 2019 we assume the following:

- CPU tasks follow the read, execute and write semantics
- A CPU task offloads workload synchronously or asynchronously to the GPU
- GPU: We assume that applications do not use unified/paged memory but only pinned memory
- GPU: We consider that each application has **one** custom stream associated with it
- GPU: We assume that there is a single copy engine to carryout host-to-device and device-to-host memory transfers
- GPU: We assume a channel per context (application), so as to have one time slice per application. Different applications might have different time slices.
- GPU: We assume no fine granular modeling of memory accesses from the execution engine. Memory accesses from the EE stage are not explicitly modeled (memory accesses are included in ticks, because copy engine also make use of Level 2 Cache). An extension is later possible
- For the CPU, priorities have to be set so as to meet deadlines. We assume that the task period is equal to its deadline.
- For simplicity, we assume that cache lines are not evicted and tasks access labels only once per activation, from memory. The participants could however override this assumption, but clearly state it out in their solution.
- Although caches are not explicitly modelled, we assume that data is always transferred as a complete cache line (64 Bytes).

3. The Challenge

The challenge consists in analytically master the complex HW/SW system (that will be available as Amalthea model) to answer the following questions:

1. Response Time Computation: Given an application consisting of a set of dependent tasks and a given mapping, calculate its end-to-end response time
    - The response time should account for the time for the copy engine to transfer data between the CPU and the GPU.
    - The response time should account for the time for the data transfers between the CPU and the shared main memory considering a read/execute/write semantic.
    - Optionally the offloading mechanism (synchronous/asynchronous) can be changed to further optimize the end-to-end latencies

- Devise a model to account for memory interference when multiple CPU cores and the GPU CE is accessing memory at the same time. The response time analyses should therefore be memory contention aware. An example of a memory interference model will be detailed later in this document.

2. Mapping Challenge: Some tasks can either be mapped to different type of CPUs (A57 or Denver Cores) or a GPU, defining a design space for exploration. Find mappings that (Pareto-) optimize the latency of the different task chains.

Note:

1. We have provided a default mapping in the Amalthea model. However this mapping is not feasible and must be changed.
2. When tasks are mapped to the GPU, the relevant time-slice parameters for each task must be provided. The current time-slice parameters are only placeholders and can be changed.

4. Application Use Case

Researchers in next-generation embedded systems agree that there is a clear need and urgency to demonstrate and bound the predictability of complex heterogeneous systems in terms of end-to-end latencies. In this section we present an overview of the application proposed for the WATERS Industrial challenge 2019.

We developed the prototype of an end-to-end autonomous driving application running on the Tegra TX2 platform[5], representative of next-generation AD/ADAS systems, because it uses state of the art algorithms in robotics/automotive, mixes different workloads with different criticality levels on the same computing platform, and exploits architectural heterogeneity by concurrently running tasks in the most suitable computational units (i.e., CPU and GPU).

The reference application provides the proper throttle, steering and brake signals to drive a vehicle through a predetermined map of waypoints. Moreover, the vehicle runs specific high-priority tasks to break and/or perform emergency maneuvers to avoid obstacles like pedestrians or bicycles, the so-called Vulnerable Road Users (VRUs). It is important to mention that the application works without using ROS[6], and all the algorithms are implemented from scratch.

We now show a brief description of the key software tasks running in the system. The green boxes represent the workloads executing on the GPU, while the blue ones execute on the CPU hosts. It is important to note that there are tasks that can be partitioned by using either a GPU or CPU implementation of it, e.g., lane detection or localization.

[5]The TX2 processor is at the base of the NVIDIA Drive board series, the state-of-the-art of high-performance embedded platform for autonomous driving and ADAS systems.
[6]http://www.ros.org/

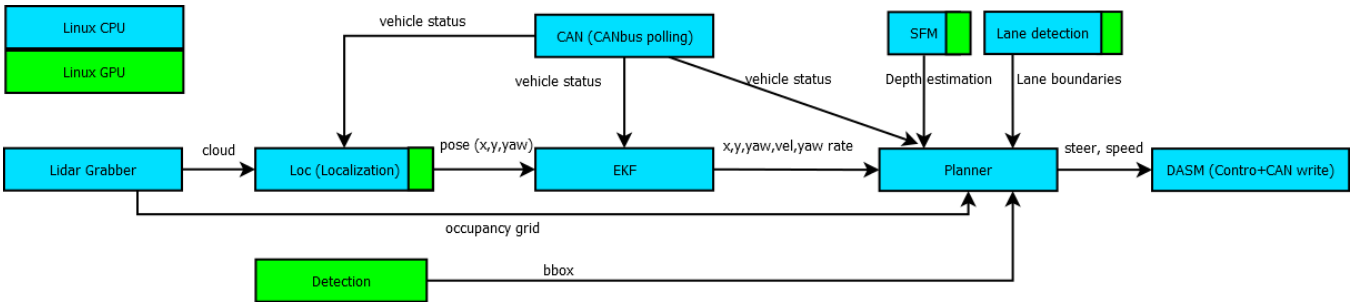


Figure 6: Application tasks

Specifically, the tasks are:

- **LIDAR Grabber.** It reads the information from the LiDAR sensor, and builds a point cloud that is shared with the *Localization* task. This task also produces a so-called *occupancy grid* that captures the free space in front of the vehicle. This information is sent to the *Path Planner* task.
- **Localization (CPU/GPU).** We implemented a *Particle Filter* localization algorithm; this method adopts a probabilistic model to determine the position of the car on the given environmental map. This position information is then merged with the motion estimation coming from the vehicle odometry, to obtain an accurate and predictive estimate of the vehicle's position. The current pose of the vehicle is sent to the *EKF* task.
- **CANbus polling.** This task snoops the key vehicle information (steer/wheel/break/acceleration status…) from the on-board CAN bus and sends it to the *Localization, Planner* and *EKF* tasks.
- **Path Planner.** The main purpose of this component is to define and follow a given trajectory. This trajectory is defined as a *spline*, that is, a line built through polynomial interpolation at times on the map that represents at each point the position and orientation that the car will have to follow. The spline can be enriched with additional information such as speed to hold, stop, priorities, etc. The planner sends the goal state of the vehicle (i.e., *target steer and speed)* to the *DASM* task that is in charge of writing the commands in the CAN line the effective steer and speed to apply.
- **DASM (Control+CAN writing).** This task computes and establishes the speed and steer that must be effectively employed from the information that is provided by the *Path Planner* task. In order to implement this node, we used two different controllers:
    - A *Pure Pursuit* controller calculates the steering angle to follow the given trajectory. The operating principle is based on a target point placed on the trajectory to be followed at a distance that varies from the vehicle depending on the speed, the angle between the vehicle and the target determines the steering angle of the vehicle.
    - Then, we use a PID controller to follow the speed profile defined on the trajectory to be followed.

  Finally, the commands are sent via CAN bus to the engine control unit in order to perform the final actuation.

- **EKF.** The *Extended Kalman Filter* is the nonlinear version of the Kalman filter method. This algorithm estimates the poses of the ego vehicle and objects repeating two stages: prediction and correction. The data produced is a matrix that corresponds to the object and ego vehicle estimation and it is sent to the *Path Planner.*

- **Lane detection (CPU/GPU).** This task identifies the lane boundaries of the road, by doing so it is possible to provide an accurate location of the road boundaries and the shape of each lane. The output of this task is a matrix of points that represent the lane boundaries within the road.
- **Detection (GPU).** This task is responsible of detecting and classifying the objects in the road. It uses a machine learning approach, with an optimized version of YOLOV2[7,8] for the TX2 platform that exploits CUDNN and TENSORRT. In this context, we re-trained the neural network in order to extend the classification classes of the baseline neural network, in this way, we support the following categories: *Pedestrians, Cars, Trucks, Bus, Motorbikes, Bicycles, Riders, Traffic lights, Traffic signs*. All the objects detected are visualized and the information produced is sent to the *Planner* task.
- **Structure-From-Motion (SFM) (CPU/GPU).** Structure-From-Motion is a method for estimating 3-D structures (depth) from vehicle motion and sequences of 2-D images. This task returns a matrix of points representing the distance with respect the objects of the image.

Please note that each node of the task graph shown in figure 6 is modelled as Liu and Layland tasks (please feel free to enrich this task model). This implies that each node is characterized by period, execution needs and a deadline. The dependencies between tasks are "soft": this means that each task is independently activated once per period and once it finishes its execution, it places the output data into a buffer in such a way that the following task can extract from this intermediate buffer the most recent data produced and work on it.

The only exception to this is represented by the pre and post processing phases of GPU tasks as described in section 5.2 and 5.3 which have a strict execution order.

Please note that not every detail is completely specified: feel free to ask for more details in the forum or to integrate what do you think is missing with reasonable and motivated assumptions.

5. Model Description in Amalthea

For describing the system model, we use the Amalthea meta-model version 0.9.3. Amalthea is organized within different sub models like software, hardware, mapping, stimuli, etc.

A complete documentation about the Amalthea version can be found on the official Amalthea Eclipse webpage[9,10]. In the following subsections, different modeling concepts are explained that are used for describing the challenge introduced earlier.

5.1. CPU memory access semantics

We assume that tasks executing on the CPU follow a read/execute/write semantic. In this model, the execution is decoupled into three different phases: a read phase where the data is fetched from memory, an execution phase, where the task performs pure computation and a write phase, when the core writes the computed result back to memory.

Memory accesses are usually represented as label accesses in Amalthea (within a runnable). The size of the label as well as the operation itself (read or write), is specified in the software model. The mapping to memory is specified in the mapping model, while the access latencies/data rates and access path can be specified in the hardware model. Figure 7 shows an example in the challenge model

[7] You only look once (YOLO) is a state-of-the-art open-source, real-time object detection system.
[8] tkDNN: https://github.com/ceccocats/tkDNN
[9] https://www.eclipse.org/app4mc/documentation/
[10] The User Guide contains a timing primer on Amalthea (Concepts → Timing in Amalthea)

for a Runnable within a Task following the read (label access to *l1*) → execute (Ticks) → write (label access to *offload1*) policy.

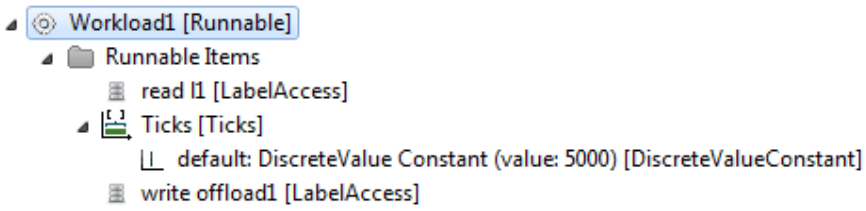


Figure 7: Runnable example for a CPU

Then the total execution time of a task is the sum of the

- Label Read access time
- Core execution time = Ticks / Frequency of the processing unit (CPU /GPU)
- Label Write access time

The label read and write access time is a function of the label size, read/write latency and the frequency of the processing unit. Then

- Read access time = (Label size /Cache line size) * (read latency / Frequency of the processing unit)

And likewise for the write access time. Note that due to simplicity, read and write latencies are always the same (we ignore store buffers, etc).

The read and write latencies are specified for every processing unit. In the model it can be founder under Processing Unit -> Access Element.

5.2. GPU Modeling

A GPU can be represented by a processing unit within the Amalthea hardware model. Due to the assumption that every application consists of one stream, it is not necessary to separate the copy engine from the execution engine as this essentially implies that interleaving between copy and execution phases cannot occur. For the abstract modeling of a GPU, we do not model the internal architecture like thread blocks, or the internal memory hierarchy explicitly. Due to its complexity, we also abstract the GPU cache system, this means that the explicit copy operations are executed (see Section 5.3) but all data accesses during the execution on the graphics/compute engine are already accounted in the “ticks” specified in the runnable.

5.3. GPU Offloading

In Amalthea parallelism is expressed on task level, this means all items (e.g. runnables) within a task are executed sequentially. Therefore, in case a workload is offloaded to the GPU, this workload is modelled as a separate task, which is shown in Figure 8.

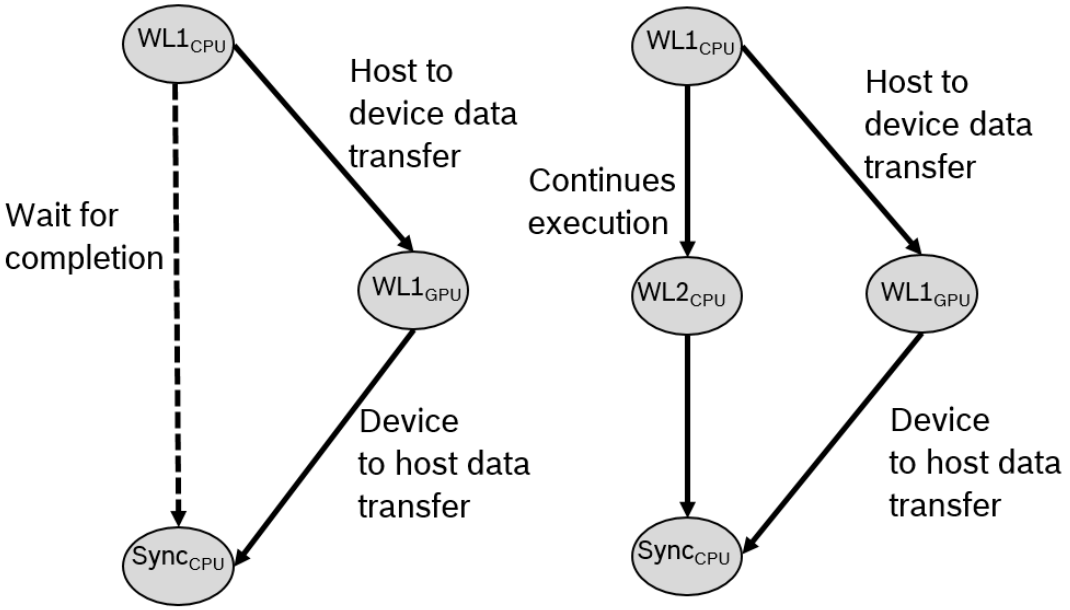


Figure 8: Modelling sync & async offloading with special copy tasks

The offloading requires data to be transferred from the CPU memory region to the GPU memory region (explained in Section 2.4).

This is accomplished by separating this offloading in three different runnables (1: host to device copy operation, 2: execution, 3: device to host copy operation). Figure 9 shows an example with three runnables and the task calling the runnables, like it is used in the provided model.

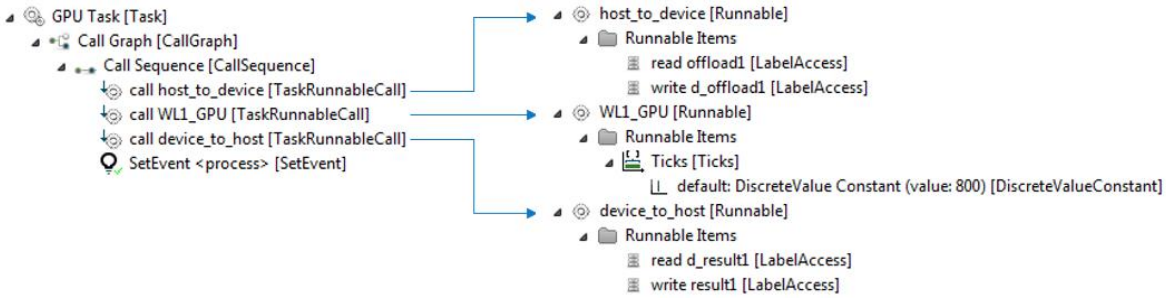


Figure 9: GPU workload example 3 runnables

In addition, the offloading can be performed in *synchronous* or *asynchronous* manner. Figure 8 shows the two different mechanisms.

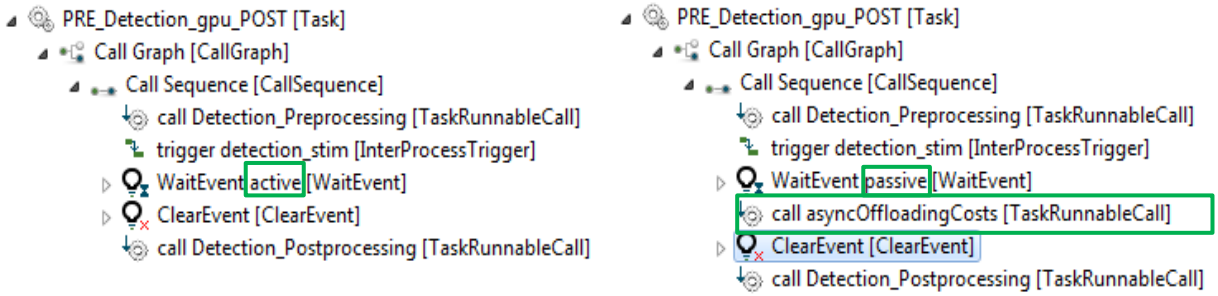


Figure 10: Synchronous offloading (left) and asynchronous offloading (right)

The workload $WL1_{CPU}$ is executed on a CPU. $WL1_{CPU}$ contains label accesses or ticks (to specify the computational cost).

- In the synchronous case, the CPU task offloads the work to the GPU, and then **actively waits** for the execution on the GPU to be completed, the resulting data is copied back from the GPU to the CPU and the corresponding event *SetEvent*[11] is set by the GPU task (see Figure 10).
  In this case, the latency between the end of the GPU kernel and the start of the postprocessing phase is 0 (**wait events are active**).
- In the asynchronous case, after the CPU task offloads the work to the GPU and goes into a **passive waiting mode**: this implies that the scheduler can schedule other workloads on the CPU before the completion of the GPU workload, specified by the "Clear Event" event. However, the latency between of the end of the GPU kernel and the start of the postprocessing phase is equal to the execution time of the "***AsyncOffloadingCosts***" runnable (**wait events are passive**) as seen in Figure 10.

Note that the "asyncOffloadingCosts" runnable is present in the model, but called by any task since we have only modelled the Synchronous Offloading case. However in case an Asynchronous Offloading has to be modelled, please insert this runnable as shown in Figure 10 and factor in the penalty in the overall response time.

The concrete modeling is shown in Figure 10. The offloading is triggered with the *GPU_offloading* inter process trigger within the task mapped to a CPU. This trigger activates a task which is mapped to a GPU and contains $WL1_{GPU}$. On the left side of the figure, the synchronous case is modeled by directly entering the *WaitEvent*. On the right side, in the asynchronous case, another workload (*WL2_CPU*) is executed before the *WaitEvent* is entered. The event itself is set in the task mapped to the GPU after finishing $WL1_{GPU}$ (see *SetEvent* in Figure 9). In both scenarios, the *ClearEvent* resets the synchronization barrier.

5.4. GPU Scheduling

For the GPU Scheduling, we assume one combined scheduler for the GPU (instead of having a separate scheduler for the copy engine and the execution engine), this is possible due to the assumption that every application has just one stream (currently represented with one task executed on the GPU). Therefore, given an application, the time slice allocated to its channel is used for both copying and execution.

We do not explicitly model channels in Amalthea but associate one GPU task to each application. These tasks are scheduled by the custom time slice scheduler of the GPU with minimum and maximum time slice of 1 ms and 500 ms respectively. These attributes are set in the Amalthea model under Operating System -> GPU_Cluster -> GPU_Sched ->User Specific Scheduling Algorithm. The scheduling policy is explained in Section 2.3.

The aforementioned weighted round robin scheduler policy is not directly supported in Amalthea, therefore a custom scheduler is used. The individual time slices for the different tasks can be specified in the mapping model when mapping a task to a scheduler "TaskAllocation". An example is shown in Figure 11.

[11] Please note that in the Amalthea model the mechanisms *wait*, *set,* and *clear event* originated from OSEK concepts, and are thus currently called OSEvents. In future Amalthea releases this concept will be generalized.

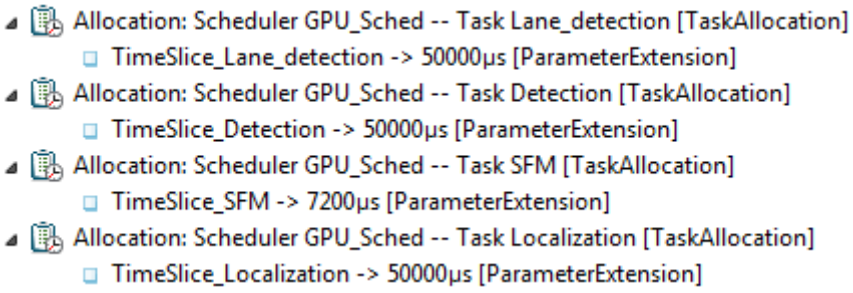


Figure 11: Specifying individual time slices for tasks

5.5. Mapping dependent resource consumption

One of the challenges is to optimize end-to-end latencies by changing the task mapping of the given model. Figure 12 shows an Amalthea modelling example, which includes multiple tick values for different mapping targets: e.g. *workload WL3* requires a default of 6000 ticks when mapped to the Cortex A57, and 3000 ticks when mapped to the GPU. Note this entries also implicitly define the possible mapping targets.

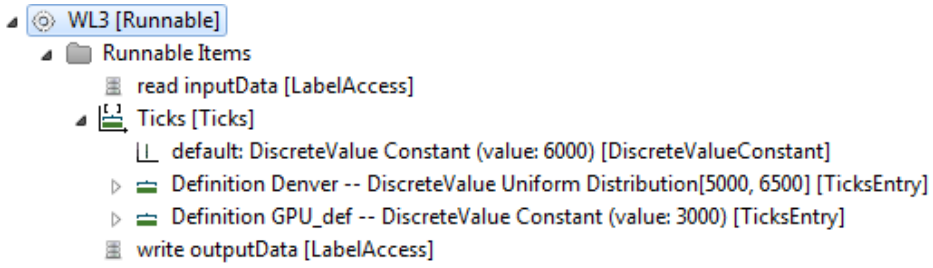


Figure 12: Extended ticks in Amalthea

Appendix

1. Transformation from a GPU Task Mapping to a CPU Task Mapping

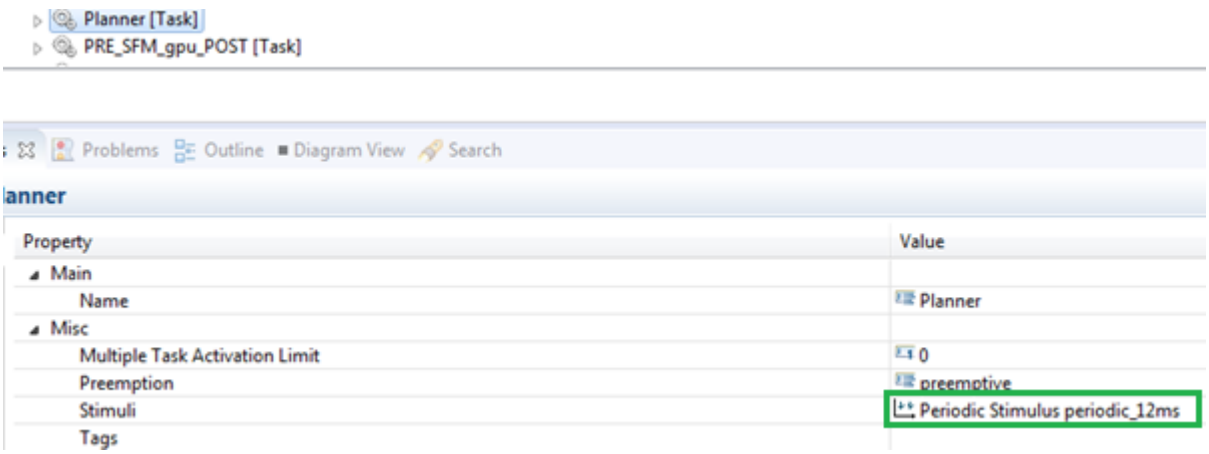
Three tasks that can be mapped either to a GPU or a CPU this tasks are **SFM**, **Localization**, and **Lane_detection**. In the provided model all cores who can run on the GPU are mapped to the GPU because in this case additional information like pre and post processing are necessary. The offloading from a CPU to the GPU is explicitly modelled with an additional task. This means whenever a remapping is done for one of the three mentioned tasks (SFM, Localization and Lane detection) the model has to be changed.

We provide a second model for just the three tasks in case they are mapped to a CPU instead of the GPU.

Example with the SFM task for changing the mapping from GPU to CPU:

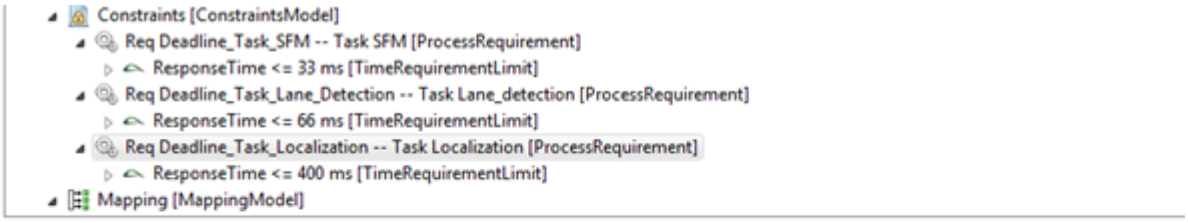
- Task *PRE_SFM_gpu_POST* has to be removed (at least trigger hast to be removed)

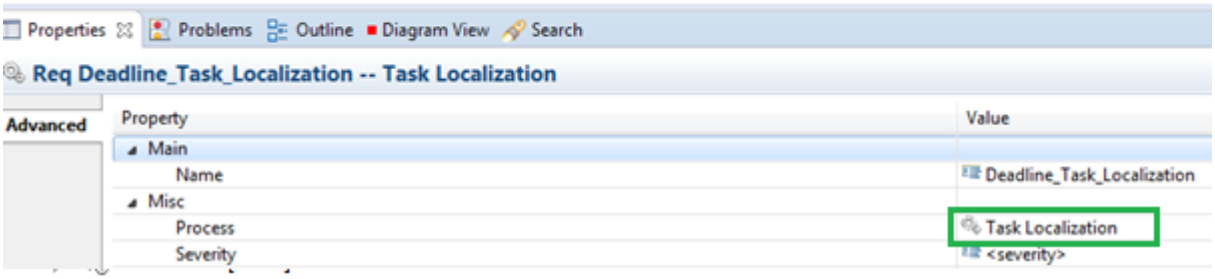
- Task trigger in the properties of the *SFM* task has to be changed from inter process trigger to the periodic trigger (*periodic_33ms*) former trigger of the *PRE_SFM_gpu_POST* task.



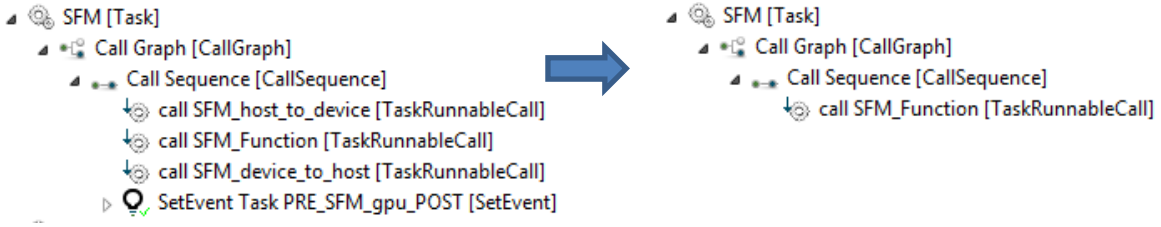
- Mapping in the mapping model for the SFM task is changed to a specific CPU (also updating the priority based on the period)

- Constraint *Deadline_Task_SFM* in the constraint model is changed for the *PRE_SFM_gpu_POST* task to the SFM task (value stays the same → 33ms)





- In the *SFM* task the runnable calls for *SFM_host _to_device* and *SFM_device_to_host* and the SetEvent element are removed (the runnables are not needed anymore)



- In the runnable *SFM_Function* the label accesses are added like in the additional model we provided (*CPU_Mapping_Changes.amxmi*).

For remapping a task from an A57 core to Denver core just the mapping in the mapping model have to be changed and another tick value and time for memory accesses have to be considered.

2. Analytical Model of Memory Contention

2.1 Modelling Memory Contention

As an addition to the challenge, we ask the participants to formulate a memory contention model when more than one CPU core and/or the GPU is accessing memory at the same time. The proposed modelling should consider the following factors.

- A task mapped to run on the GPU, needs offloading data that is acquired through the copy engine (GPU CE)
- A GPU kernel outputs data that needs to be copied back to the host.
- CPU tasks are modeled as with a Read/Compute/Write semantic, with "Read" and "Write" being 100% memory bounded operations
- GPU CE, A57 and Denver have significantly different memory bandwidth, latencies and sensitivity to memory interference.
- The size of buffers to be read and to written is known in advance and is fixed for every instance of the periodic task.

We then need to derive the read/write latencies when more than one CPU core is accessing main memory at the same time. We present an example memory model below and the authors could build their analysis on this or present a refined contention model.

2.2 Example Memory Model

We consider the following work that measures memory impact on interference:

https://ieeexplore.ieee.org/stamp/stamp ... er=8247615
http://hercules2020.eu/wp-content/uploa ... tforms.pdf

The idea is that the length of memory phases (read and write) depends on how many other memory controller clients are accessing **main** memory at the same time. The model accounts for increasing latencies for GPU CE activity during the observed time window. The method to compute memory contention for CPU and GPU tasks is presented as below.

2.2.1 Modelling Memory Contention for CPU Tasks

Lat(CPUtype,cacheLine)[ns] = baseline(CPUtype) + K(CPUtype)*#C + sGPU(CPUtype)*bGPU

with:

- Lat(x,y) = time needed to read or write a cacheLine (64B) from main memory to CPU registers.
- CPUtype = Observed CPU core; A57 or Denver
- baseline = time taken to read or write a cacheLine (64B) from main memory to CPU registers in isolation (no interference).
  baseline(A57) = 20 ns
  baseline(Denver) = 8 ns
- K(CPUtype) = increase in latency operated by a single interfering core.
  - Do note: it does not matter if the interfering core is denver or A57: this number only depends on the observed CPU core (CPUtype)
    K(A57) = 20 ns
    K(Denver) = 2 ns

- #C = number of interfering cores. Range from 0 to 5 (as one core is the observed CPU core. 0 means no interference from other CPUs)
  - sGPU = sensibility to GPU CE activity. This represents an increase in latencies if the GPU is performing operations on the copy engine.
    sGPU(A57) = 100 ns
    sGPU(Denver) = 20 ns
- bGPU = boolean. 1: if the GPU is operating the copy engine, 0 otherwise.

2.2.2 Modelling Memory Contention for GPU Tasks

From the GPU side, we only consider interference from the Copy Engine data movements (modeled in Amalthea as "runnables"). A GPU CE data movement is a 100% memory bound runnable. Every memory access is modeled as sequential access.

Lat(memcpy,64B) = GPUbaseline + 0.5*#C

with

- Lat(memcpy,64B) = Time taken to transfer 64B using the copy engine (cudaMemcpy)
- GPUbaseline = 3 ns. Time taken to transfer 64B using the copy engine with no interfering CPUs .

Each core active in the same time window of the CE operation increases the baseline by half a nanosecond.

2.3 Numerical Example

A task mapped on a A57 core has a memory footprint (read) of 128B. If no other CPU is accessing memory (#C=0) and if the GPU CE is idle (bGPU=0), then the time necessary to perform the read operation of the working set size is:

128/cacheLine*Lat(A57,cacheLine) = 2*20 = 40 ns .

If one core is active (does not matter if Denver or A57):

128/cacheLine*Lat(A57,cacheLine) = 2*(20 + 20*1 + 0) = 80 ns.

And this would increase to 280 ns if the GPU CE was active for the whole time of this memory phase.

List of Figures