PasTiS: building an NVIDIA Pascal GPU simulator for embedded AI applications

Michaël Adalbert, Thomas Carle, Christine Rochange

PasTiS: building an NVIDIA Pascal GPU simulator for embedded AI applications

Michaël Adalbert*†, Thomas Carle†, Christine Rochange†
*IRT SystemX, Palaiseau, France
†IRIT - Univ. Toulouse III - CNRS, Toulouse, France, name.surname@irit.fr

Index Terms—**GPU, cycle-accurate simulator, timing analysis**

***Abstract*—We present PasTiS, a simulator for the NVIDIA Pascal GPU architecture family, with a focus on timing simulation. PasTiS supports a subset of the Pascal ISA, sufficient to simulate the execution of neural networks. We present this subset, as well as the underlying microarchitecture that we modelled using information available from NVIDIA, from scientific publications, and from our own experiments. We demonstrate the precision of the simulator by comparing it to measurements on the NVIDIA Jetson TX2 development board, on neural network applications.**

I. INTRODUCTION

Real-time systems are increasingly embedding machine learning software which requires a huge computing power. For example, the control systems of autonomous vehicles rely on neural networks (NNs) to detect roads and objects, compute trajectories and plan the actions to be performed. These algorithms are computation-intensive and inherently parallel, which drives the industry to adopt massively parallel hardware such as many-core and GPU accelerators. In particular, GPUs have received a lot of attention these last years, both from the industry and from the real-time research community.

Scheduling tasks in a timing-critical system, so as to ensure that they will meet their timing constraints, requires being able to determine their respective worst-case execution time (WCET). Various approaches to WCET analysis exist and are based on static analysis techniques and/or measurements [10]. The estimated WCET can then be deterministic (i.e. expressed as a single upper bound) or probabilistic (i.e. several WCET values are produced, each associated to a probability of being exceeded) [9]. In this paper, we focus on systems where strict upper bounds on execution times are needed, and thus consider static WCET analysis approaches.

Static WCET analysis aims at determining invariants on the code of the task under analysis. Some invariants are related to the software (e.g. loop bounds), others to the state of the hardware (processor, cache memories, etc.). They are all used to build an integer linear program (ILP) that maximizes the execution time of the task over all the possible paths in the control flow graph (CFG) [17]. OTAWA is an open-source framework that offers many built-in facilities to generate WCET analysis tools [4].

This work was partially supported by the ANR LabEx CIMI (grant ANR-11-LABX-0040) within the French State Programme "Investissements d'Avenir."

Computing hardware-related invariants requires modeling the behaviour of the computing platform. This is usually done manually by translating the knowledge we have of the hardware (from documentation provided by the processor manufacturer or designer) into a formal model, although automatic translation from a VHDL specification of the processor (when available) has also been considered [24].

However, most of existing work considers CPU-only platforms. The very specific execution model of GPUs (Single Instruction Multiple Threads - SIMT) requires a substantial revisiting of hardware models. First, the lockstep execution of batches (*warps*) of threads requires rethinking the concept of basic block and instruction sequence due to possible branch divergence: all the threads in a warp do not necessarily follow the same control flow. Second, GPUs implement hardware scheduling schemes for warps and blocks of threads that must be accounted for in WCET estimations. Third, their memory system is noticeably different from that of standard CPUs and requires specific analyses.

The closed nature of most GPUs and the lack of official documentation on their micro-architecture have slowed down the understanding and modeling of their micro-architectural behaviour, which plays a crucial role in the execution time. These reasons explain that so far, no decent (industrial or academic) WCET analyzer for GPU-accelerated code is available.

As part of the French national *Confiance.ai* [1] program that brings together industry and academic researchers to build trustworthy Artificial Intelligence, we ambition to extend static WCET analysis techniques to GPUs. The work presented in this paper was mainly supported by the Labex CIMI[2] through the AVATAr project, and is a first step towards this objective: we show how we were able to conduct experiments to uncover some of the execution mechanisms and hardware parameters of an NVIDIA Pascal GPU and how we have built a cycle-level simulator that is able to run CUDA programs. We compare the execution times evaluated by the simulator to those measured on a Jetson TX2 board and demonstrate the accuracy of our model.

The insights provided in the paper can be used by industrial actors to assess the viability of using a GPU in their embedded systems: they give a clearer view of how a GPU program is actually executed and of the specificity with respect to CPU

[1] https://www.confiance.ai/en/
[2] https://cimi.univ-toulouse.fr/en/

execution. Although we focus on a particular NVIDIA GPU, the general mechanisms that we describe are common to all GPUs.

Contributions: In this paper, we present how we have proceeded to understand the behavior of an NVIDIA Pascal GPU and how we have used this knowledge to develop a cycle-level simulator called PasTiS (Pascal Timing Simulator).

Section II describes the general organization and execution model of GPUs, with a focus on our target (NVIDIA Pascal architecture). In Section III, we detail two key aspects of the GPU behaviour (thread divergence and accesses to the shared memory), and how we proceeded to understand them. We then introduce the PasTiS simulator in Section IV, and evaluate its performance in Section V. Related work is discussed in Section VI. Section VII concludes the paper.

II. GPU ORGANIZATION AND EXECUTION MODEL

In this section we present the global organization and execution model of GPUs. We borrow the NVIDIA terminology which is widely accepted in the community, but similar concepts exist in GPUs from other manufacturers.

A. Heterogeneous computation

At the highest level, a GPU is an accelerator on which a CPU can offload functions called *kernels*. A kernel is specified in a particular language (or a language extension, such as CUDA or OpenCL) that enables the description of parallel computation. The offloading of a kernel to a GPU generates threads that all execute the same code. The number of threads is specified by the programmer, as a number of *blocks* and a number of threads per block, and should be as high as possible in order to fully benefit from the GPU's resources. This is illustrated in Figure 3. In this example, a CPU function (*lines 2-11*) modifies each element in an array, depending on its initial value. Instead of processing elements sequentially, it invokes a GPU kernel (*line 7*) for a number of threads equal to the array size (`n × BLK`). The code of the kernel is given on *lines 12-18*. Each thread executing this code determines its identifier (*line 13*) – a value between 0 and `n × BLK`, and according to the initial value of the element (*line 14*), it computes its new value (*lines 15* and *17*). The CPU and the GPU have distinct main memories and the GPU cannot access the CPU memory. For this reason, the CPU function has to allocate space in the GPU memory (*line 4*) and to transfer input data from the CPU memory to the GPU memory (*line 5*) and output data from the GPU memory to the CPU memory (*line 8*).

B. General organization

On the hardware side, a GPU is a collection of clusters called *Streaming Multiprocessors* (SMs). Figure 1.a shows that the NVIDIA Pascal GPU of the Jetson TX2 contains two SMs, that share an L2 cache. Each SM contains in turn four processing blocks, denoted SMPs, that share an L1 instruction cache and two L1 data caches (each shared by two SMPs), as well as a fast multi-banked memory referred to as the *shared memory* (Figure 1.b). An SMP includes 32 *Cuda Cores* (CCs) that perform ALU and floating-point (32- and 16-bit) operations, a 64-bit FP unit, 8 Special Function Units and 8 Load/Store Units (Figure 1.c). It is then able to execute several operations in parallel, e.g. 32 integer instructions or 8 memory loads. In addition, the GPU contains resources to host thousands of active threads so that context switching is extremely fast.

When a kernel is offloaded to a GPU, each block of threads (e.g. 1024 threads in the example of Figure 3) is mapped to one SM. This mapping is performed by the hardware, following a heuristic that tries to maximize the use of the SMs [21].

C. Execution model

At the lowest level, GPUs implement the Single Instruction Multiple Threads (SIMT) execution model, which can be seen as a mix between simultaneous multithreading and the SIMD[3] model. A block of threads is organized into fixed pools of 32 threads called *warps*[4]. All the threads inside a warp are executed in lockstep: in a given clock cycle, they all execute the same instruction. This reduces the complexity of the instruction fetch and decoding logic since these operations are shared between the 32 threads of a warp.

Warps are the smallest schedulable entities in a GPU. An SMP contains an instruction buffer for each active warp, which stores the next instructions to be executed by the warp. At each execution cycle, a hardware warp scheduler is responsible for electing a warp for execution among those that are ready, following a given scheduling policy [19]. A warp is ready for execution when all data dependencies have been resolved (which is checked using a scoreboard, as explained in Section II-D), when its next instruction is available in its instruction buffer and when the required functional units are available. The instruction to be executed by the elected warp is sent to a dispatch unit that pushes the instruction to the required functional units. Each SMP has two dispatch units: one for memory operations (which target LSUs) and one for the other instructions. The full processing of an instruction is depicted in Figure 2.

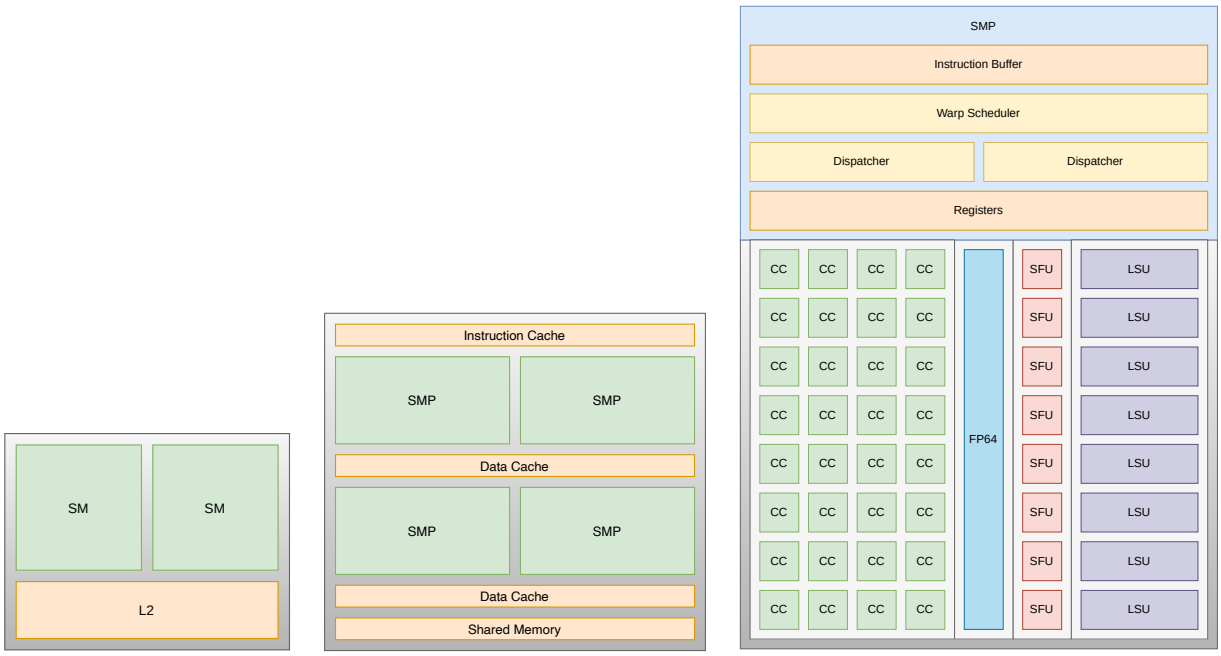
As long as all threads inside a warp agree on the control flow (i.e. they follow the same direction at conditional branches), the SIMT execution scheme is straightforward. However, when threads within the same warp disagree on whether or not to take a branch, both paths are executed *in sequence* one after the other, with only one part of the threads being active on each path. This phenomenon is known as *thread divergence* [8].We explain in Section III-A how thread divergence is handled in Pascal GPUs.

D. Scoreboard update and scheduling instructions

Recent NVIDIA GPUs handle pipeline hazards using a software programmed scoreboard: in a Pascal GPU program, the compiler inserts so-called *scheduling instructions* [11]

[3]Single Instruction Multiple Data

[4]The number of threads in a warp is not the same in all GPUs depending on the vendors. In NVIDIA GPUs, warps are always composed of 32 threads.



(a) GPU (b) SM (c) SMP

Fig. 1. Architecture of a Pascal GPU

between each group of three successive instructions. These are responsible for updating the scoreboard with information needed to enforce data dependencies (minimum instruction latencies, setup of local fences).

E. Memory hierarchy

All the SMs share a *global memory* that can be used by their threads to communicate between SMs, and a constant memory which stores constants. The global memory is also used to communicate between the CPU and the GPU through the use of a hardware *copy engine* that transfers data and instructions between the CPU and GPU memory spaces. Threads can also access a private *local memory*, which is in practice an area of the global memory. The global, constant and local memories are accessed through a cache hierarchy composed of an L2 cache, shared by all the SMs, and L1 caches, local to each SM. The memory hierarchy of the Pascal GPU is displayed in Figure 4.

In addition, each SM features a so-called *shared memory*. In the GPUs terminology, shared memory refers to a fast memory local to each SM, that is shared by threads that belong to the same block. NVIDIA reports a 100x lower latency for shared memory accesses compared to uncached global memory accesses, making it a key factor in the acceleration of kernel execution. It is implemented as an interleaved multi-banked SRAM with 32 banks storing 32-bit words. In the ideal case, threads of a given warp access either words from different banks or the same word from a given bank. The hardware is optimized to serve such requests with minimal latency and maximal throughput by grouping them into a single transaction. However, whenever two or more threads from the same warp try to access different addresses in the same bank at the same time, this results in a conflict. The bank then serves each request in sequence as part of a separated transaction, and all the threads in the warp wait until all requests have been served before executing the next instruction. This obviously degrades performance.

III. Reverse engineering the Pascal GPU execution

A. Thread divergence

Thread divergence occurs when threads within a warp do not all follow the same direction after a conditional branch. For example, in the code in Figure 3, threads execute either *line 15* or *line 17* depending on their own evaluation of the condition on *line 14*. As mentioned earlier, the SIMT execution model does not allow threads that belong to the same warp to execute different instructions. As a result, each path (i.e. *line 15* or *line 17*) is executed one after the other one, and each thread is active along one of these paths only.

According to [2], thread divergence is handled using a hardware mask mechanism that temporarily deactivates the threads that must not execute one path. In order to deal with nested conditional branches, a stack (called SIMT stack in the remainder of the paper) holds the activation masks, along with some additional information. In order to better understand the behavior of the SIMT stack, we read a NVIDIA patent dedicated to this mechanism [20]. In this section we present a model of the divergence handling mechanisms of the Pascal GPU, based on this patent and on experiments that we conducted in order to verify and clarify the behaviour of the GPU on conditional branches.

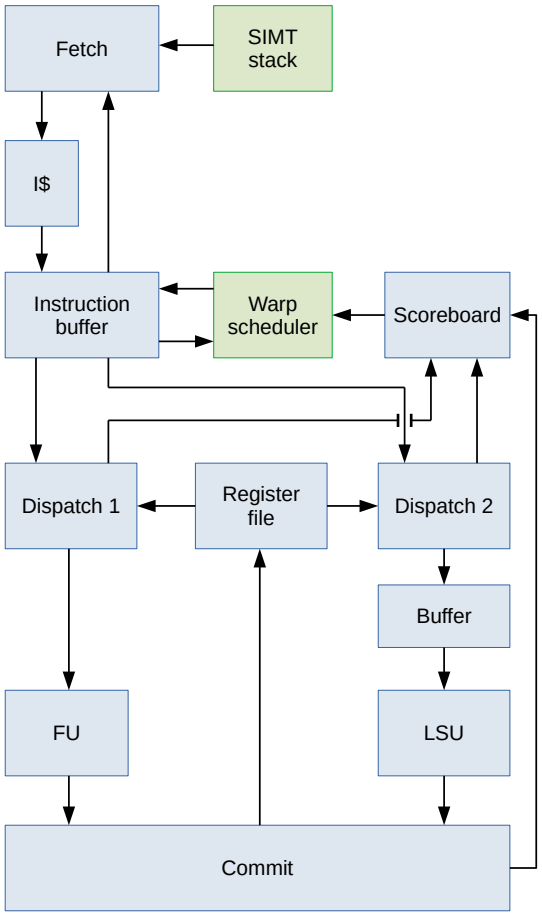


Fig. 2. Detailed view of the modeled elements for one SMP in the PasTiS simulator. In blue, the elements related to the non-functional semantics of instructions. In green, the elements related to the warp-level semantics.

```
1   #define BLK 1024
2   void fun(int *a, int n){
3       int *d_a;
4       cudaMalloc((void **)&d_a,n*BLK*sizeof(int));
5       cudaMemcpy(d_a, a, n*BLK*sizeof(int),
6                  cudaMemcpyHostToDevice);
7       kern<<<n,BLK>>>(d_a, n);
8       cudaMemcpy(a, d_a, n*BLK*sizeof(int),
9                  cudaMemcpyDeviceToHost);
10      cudaFree(d_a);
11  }
12  __global__ void kern(int *t, int n){
13      int tid=blockIdx.x*blockDim.x+threadIdx.x;
14      if (t[tid] < 0)
15          t[tid] = 0;
16      else
17          t[tid] *= 2;
18  }
19
```

Fig. 3. Example Cuda program

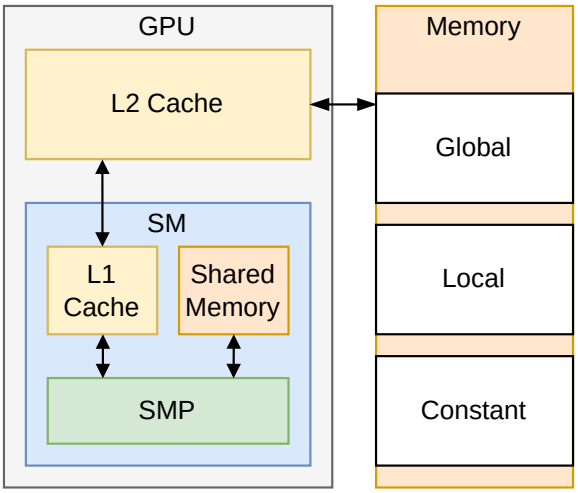


Fig. 4. Memory hierarchy

When a kernel is launched, a stack is allocated for each corresponding warp. In the stacks, a 32-bit mask is stored along with the address of the next instruction to execute (next program counter or *npc*) and of the address of the instruction at which the threads must wait for reconvergence (reconvergence program counter or *rpc*). Each stack is initialized with an entry composed of: a mask in which all the threads in the warp are active, the start address of the program as *npc* and the last address in memory as *rpc* (note that it does nos need to be a valid instruction address). The left part of Figure 5 shows the state of the SIMT stack for a warp executing the kernel of Figure 3: the next instruction to be executed is at address `0x8`, the end of the program is at address `0xFFFF` and all the threads of the warp are currently active. The GPU handles divergence and reconvergence through the use of dedicated instructions which are automatically inserted in the code by the compiler. In practice these instructions are responsible for modifying the SIMT stack of their warp. The `SSY` and `PBK` instructions prepare the stack for a possible divergence; they contain the reconvergence address. When `SSY @addr` or `PBK @addr` is executed, the top entry of the stack is popped. A new entry is pushed to handle the execution after reconvergence: the *npc* is the reconvergence address (`@addr`), and the *rpc* and the mask are copied from the popped entry. A second new entry is then pushed to handle the potentially diverging portion of code: the *npc* is the actual next instruction (the *npc* of the popped entry + 8, since instructions are encoded on 64 bits), the *rpc* is the reconvergence address (`@addr`) and the mask is the same as in the popped entry. In the example of Figure 5, we consider that the instruction at address `0x8` is a `SSY` or `PBK` instruction that prepares the stack for a future conditional branch (corresponding to the *if-else* construct in the code of Figure 3), with a reconvergence of the control flow at address `0x70`. The stack is updated (in the right part) with two entries. The bottom entry will be used after reconvergence: the *npc* is the address of reconvergence of the control flow (`0x70`). The top entry is used to let all active threads execute the next instruction (`0x10`) naturally. Note that these instructions do not create divergence but instead prepare the stack for a possible upcoming divergence, which is why the mask remains unchanged at this point. In our experiments, we encountered

the SSY instruction before branch instructions that correspond to either an IF or a SWITCH construct, and the PBK instruction before branch instructions corresponding to a loop, although the official documentation from NVIDIA indicates that PBK instructions are not supported by Pascal (and next-generation) GPUs.

Actual divergence happens when a branch (BRA @addr) instruction is executed conditionally by only a subset of the threads of a warp. When this happens, the top entry of the corresponding stack is popped. Two new entries are then pushed:

- the first pushed entry concerns the threads which do not take the branch: the *rpc* is the same as in the popped entry, the *npc* corresponds to the next instruction in the code (i.e. the current *npc* + 8) and the mask activates only the threads that do not take the branch (the ones for which the condition is false);
- the second entry has the target address of the branch as *npc* and the same *rpc* as the popped entry; its mask activates only the threads that take the branch (the ones for which the condition is true).

As a consequence, the GPU first executes the threads that take the branch, until they reach a reconvergence instruction which is added by the compiler: SYNC of BRK (depending on whether a SSY or a PBK was executed before). This process is illustrated in Figure 6. In this example, the threads reach a BRA 0x48 instruction at address 0x10. We assume that threads 0 and 1 of the considered warp execute the *else* part of the code because the elements they access in table t are positive, while the rest of the threads execute the *if* part. As a consequence, only threads 0 and 1 take the branch instruction while all other threads in the warp continue in sequence. As displayed in the right part of the figure, the top entry is replaced by two new entries: one for threads 0 and 1 (activation mask 0xC0000000) and one for the rest of the threads (mask 0x3FFFFFFF). To the best of our knowledge, no documentation explicitly describes the order in which the entries are pushed on the stack when divergence occurs (and thus the corresponding execution order of the branches). In our experiments, the first threads to execute are always the ones taking the branch.

The reconvergence instructions pop the top entry from the stack, as illustrated in Figure 7. In the example, threads 0 and 1 reach a SYNC (or BRK) instruction: their entry is popped from the stack. The GPU then resumes the execution with the group of threads active in the mask of the new entry at the top of the stack: the threads that do not take the branch. When they reach a SYNC (or BRK) instruction, their corresponding entry is popped from the stack: the reconvergence is done and the execution flow resumes at the reconvergence address (which is the *npc* of the entry at the top of the stack at this point).

B. Shared memory accesses

We needed to characterize the timing behavior of the shared memory in order to implement it in the simulator. The major difficulty was to derive the number of transactions generated by the hardware. It depends on (i) the memory addresses accessed by threads in a warp, and (ii) the size of the transferred data. To determine this number, we wrote a microbenchmark in which we control the size of the transferred data and the target memory addresses, so as to tune the number of conflicts in the memory banks. We then executed the benchmark on the Jetson TX2 board and observed the number of transactions using the NVIDIA nvprof profiling tool.

Before:

0x8	0xFFFF	0xFFFFFFFF
npc	*rpc*	*mask*

After:

0x10	0x70	0xFFFFFFFF
0x70	0xFFFF	0xFFFFFFFF
npc	*rpc*	*mask*

Fig. 5. SIMT stack when executing SSY/PBK 0x70

Before:

0x10	0x70	0xFFFFFFFF
0x70	0xFFFF	0xFFFFFFFF
npc	*rpc*	*mask*

After:

0x48	0x70	0xC0000000
0x18	0x70	0x3FFFFFFF
0x70	0xFFFF	0xFFFFFFFF
npc	*rpc*	*mask*

Fig. 6. SIMT stack when executing BRA 0x48 by the first 2 threads

Before:

0x48	0x70	0xC0000000
0x18	0x70	0x3FFFFFFF
0x70	0xFFFF	0xFFFFFFFF
npc	*rpc*	*mask*

After:

0x18	0x70	0x3FFFFFFF
0x70	0xFFFF	0xFFFFFFFF
npc	*rpc*	*mask*

Fig. 7. SIMT stack when executing SYNC/BRK by threads 0 and 1

	size of accesses (# bits)		
mask	32	64	128
000000FF	1	2	4
0000FFFF	1	2	4
00FFFFFF	1	2	4
FFFFFFFF	1	2	4

TABLE I
NUMBER OF TRANSACTIONS FOR CONSECUTIVE ACCESSES

We report in Table I the number of observed transactions when threads in a warp make accesses to consecutive areas of the memory (e.g. thread 0 accesses a 32-bit word at address 0, thread 1 accesses a 32-bit word at address 4, etc.). Table I also displays the activation mask that we use in order to control which threads are active. For 32-bit accesses, a single

#	mask	size of accesses (# bits) 32	64	128
1	00000001	1	2	4
2	00000003	2	3	5
3	00000007	3	4	6
4	0000000F	4	5	7
5	0000001F	5	6	8
6	0000003F	6	7	9
7	0000007F	7	8	10
8	000000FF	8	9	11
9	000001FF	9	10	11
10	000003FF	10	11	12
11	000007FF	11	12	13
12	00000FFF	12	13	14
13	00001FFF	13	14	15
14	00003FFF	14	15	16
15	00007FFF	15	16	17
16	0000FFFF	16	17	18
17	0001FFFF	17	17	18
18	0003FFFF	18	18	19
19	0007FFFF	19	19	20
20	000FFFFF	20	20	21
21	001FFFFF	21	21	22
22	003FFFFF	22	22	23
23	007FFFFF	23	23	24
24	00FFFFFF	24	24	25
25	01FFFFFF	25	25	25
26	03FFFFFF	26	26	26
27	07FFFFFF	27	27	27
28	0FFFFFFF	28	28	28
29	1FFFFFFF	29	29	29
30	3FFFFFFF	30	30	30
31	7FFFFFFF	31	31	31
32	FFFFFFFF	32	32	32

TABLE II
TRANSACTIONS ISSUED WHEN ALL THREADS ARE IN CONFLICT

transaction is issued by the hardware, regardless of the number of active threads: each memory bank composing the shared memory is accessed by one and only one thread, and the transaction accounts for up to 32 non-conflicting accesses to 32-bit words. For 64-bit accesses, two transactions are issued, regardless of the number of active threads. In this setting, when more than 16 threads are active, at least one bank is accessed by two threads that request different words: this conflict is handled by issuing two transactions. However, when less than 17 threads are active and are consecutive, the accesses made by the active threads are non-conflicting, and we initially expected to measure a single transaction, as in the case of 32-bit accesses. For 128-bit accesses, four transactions are issued, regardless of the number of threads. Once again, we expected this behavior when more than 24 threads are active, but based on the number of conflicting accesses in the banks, we expected a single transaction when less than 9 (consecutive) threads are active, two transactions when between 9 and 15 consecutive threads are active, and three transactions when between 16 and 23 consecutive threads are active.

We designed a first experiment in which all active threads access different words in the same bank at the same time, and in which we vary the number of active threads from 1 to 32 and the size of the access from 32 to 128 bits. For each

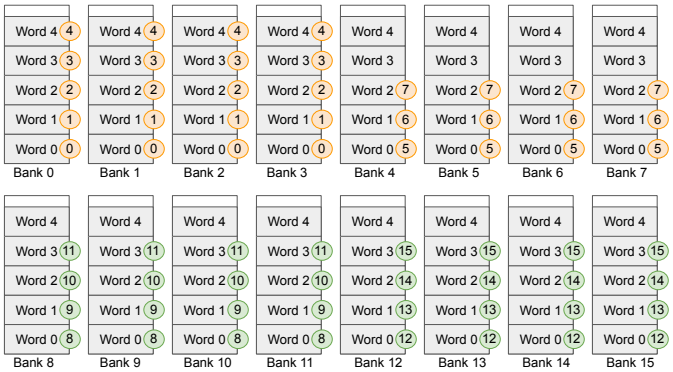


Fig. 8. Conflicts between shared memory accesses

combination of active threads and access size, we measured the number of issued transactions using `nvprof`. We discovered that threads of a warp that perform memory accesses are grouped in one pool of 32 threads when the accesses performed are 32-bit wide, in two pools of 16 (threads 0 to 15 and 16 to 31) when the accesses are 64-bit wide and in four pools of 8 (threads 0 to 7, 8 to 15, 16 to 23 and 24 to 31) when the accesses are 128-bit wide. The results of the experiment are displayed in table II. By default, one transaction is issued for each pool of threads when a warp executes a memory access. As reported in Table II we measured that even when all the threads in a given pool are inactive, a transaction is still issued for the pool. Since transactions are issued pool-wise, only threads from the same pool can generate conflicting accesses to the memory banks. We ran some additional experiments to confirm that hypothesis. In these experiments we forced conflicts to happen between threads from the same pool on two separate banks. The banks accessed by threads of different pools are not the same, in order to evaluate if the hardware tries to coalesce accesses from different groups in transactions. Figure 8 displays an example of such an experiment. For space reasons, we only display the first 5 words of the first 16 banks of the shared memory. In the example, threads 0 to 15 perform 128-bit accesses to the memory: each thread thus performs accesses to 4 consecutive banks. Since accesses are of size 128 bits, threads 0 to 7 are in a pool (shown with orange disks) while threads 8 to 15 are in another pool (shown with green disks). Threads 16 to 31 are not displayed, once again for space reasons, and without loss of generality we can assume that they are not active in the example. The threads are programmed so that in the orange pool, threads 0 to 4 are in conflict on banks 0 to 3 and threads 5 to 7 are in conflict on banks 4 to 7, and in the green pool threads 8 to 11 are in conflict and threads 12 to 15 also. Running the corresponding program on the GPU generates a total of 9 transactions.

After multiple experiments in which we varied the configuration of the conflicts, we observed that transactions were generated in order to deal with the conflicts of each pool separately, and that for each pool, the number of generated transactions corresponds to one plus the maximum number

of conflicts on any bank between threads of the pool. In the example of Figure 8, the maximum number of conflicts on any bank is 4 for the orange pool, and 3 for the green pool. In our hypothesis, the GPU is thus expected to generate $(4+1)+(3+1)=9$ transactions which is exactly what we measured. Following our observations, we derived the following formula to compute the number of transactions issued by the hardware:

$$nTrans = \sum_{i=1}^{nPools} (1 + \max_{j \in [|0,31|]} (conflict(pool_i, bank_j)))$$

where $nTrans$ is the number of issued transactions, $nPools$ is the number of thread pools (1, 2 or 4) and $conflict(p, b)$ is the number of conflicting accesses by threads of pool p on bank b.

We used the same benchmarks and measured the number of cycles for the accesses instead of the number of transactions. We could derive the following formula:

$$dur = 22 + base + 2 \times \sum_{i=1}^{nPools} (\max_{j \in [|0,31|]} (conflict(pool_i, bank_j)))$$

where dur is the duration in cycles of the execution of the memory instruction and $base$ is a constant duration which depends on the size of the accesses: 1 cycle for 32-bit accesses, 8 cycles for 64-bit accesses and 16 cycles for 128-bit accesses.

IV. The PasTiS simulator

A. Global architecture of the simulator

The simulation of a CUDA program with PasTiS mimicks a real execution shared between a CPU host and a GPU accelerator target, as illustrated in Figure 9. The CPU parts of the program (e.g. the `fun` function), are compiled and executed natively by the CPU of the machine that hosts the simulation, and the call to a GPU kernel is replaced by a call to PasTiS (which is developed in C language). The simulator code and the CPU part of the program are linked together and share the same memory space. As a result, the simulator can read and write in a memory zone accessible to the host program[5]. Before starting the cycle-level simulation, the `simulate` function stores the parameters in the constant memory and initializes special registers for each thread (e.g. with its identifiers, such as `blockIdx.x`), as expected by the GPU assembly code.

PasTiS is organized around three main modules that work together to enable a cycle-accurate simulation of GPU code:

1) an instruction set simulator: it is responsible for decoding and functionally simulating GPU binary code. This means maintaining the contents of registers and of the memory all along the execution, by emulating the semantics of the program's instructions. The instruction set simulator is agnostic of the timing aspects of the execution.

[5]PasTiS is intended to simulate and determine the duration of the execution on the GPU, but not of the operations of the copy engine.

```
1  #define BLK 1024
2  void fun(int *a, int n){
3      parameter_t par[2];
4      par[0].kind = PTR; par[0].val.ptr = a;
5      par[1].kind = INT; par[1].val.i = n;
6      simulate("kern.cubin", n, BLK, par);
7      // instead of kern<<<n,BLK>>>(a,n);
8  }
9
```

Fig. 9. Modification of the example in Figure 3 to invoke the simulator

2) a model of the warp-level execution semantics: it simulates warp selection and scheduling policies, as well as control flow and intra-warp thread divergence.
3) a model of the timing aspects of the execution: it maintains the state of the architectural components (e.g. contents of cache memories, occupation of functional units, contents of the scoreboard, etc.) and determines instruction latencies in such a way that it is able to determine the full state of the GPU after each simulated clock cycle. Thanks to this module, PasTiS is able to derive the total execution time of a kernel.

When the simulator is launched, elements modeling the memory, caches, scoreboard, registers and SIMT stacks are allocated. Then, a loop simulates the execution cycle by cycle: for each SMP, if a warp is ready for execution, the warp scheduler simulator selects the warp to execute and the instruction set simulator decodes and emulates the corresponding instruction. The latency of the instruction is computed depending on its nature, the state of the caches, and the potential memory access conflicts. Finally the state of the hardware simulator is updated according to the effect of the current instruction.

B. Instruction set simulator

The instruction set simulator is in charge of decoding the binary code of the program and emulating its execution from a functional perspective. For this purpose, we rely on GLISS [23], a tool that accepts the description of an ISA (its encoding and its semantics) and generates a library of functions that can be invoked to decode an instruction or to evaluate its impact onto the processor's state (registers and memory contents). We have described a large part of the Pascal ISA in the GLISS format.

The Pascal GPUs use the same ISA as the Maxwell generation. Unfortunately, the low-level (SASS)[6] language is not documented by NVIDIA: only the instruction names are provided, but not their encoding nor their exact semantics. We thus had to conduct reverse engineering in order to determine the missing information. The work reported in [11] was a valuable starting point for this task, which is really tedious and which we have limited to the subset of instructions that appear in our benchmark applications (in particular matrix products and

[6]PTX is a common ISA for all Nvidia GPUs, while SASS is a micro-architecture specific ISA. SASS code is generated by JIT or AOT compilation of PTX sources

a feed-forward fully-connected neural network). The Maxas tool[7] was very useful to uncover the entire encoding scheme for some of the instructions. To understand the semantics of the instructions, we carefully analyzed the mapping between source and binary code, and we could clarify uncertainties based on NVIDIA-related forums.

The method for deriving the encoding of instructions is the following:

1) compile a GPU kernel to a .cubin binary file using the nvcc compiler
2) disassemble the .cubin file using nvdisasm
3) select an instruction to investigate
4) modify the disassembled program to generate multiple instances of the instruction with varying operands and options
5) re-assemble the program using maxas
6) disassemble again, and compare the binary encodings of the instructions

The second step to building an instruction set simulator is to describe the semantics of the instructions. Once again some reverse engineering had to be performed for this step. To do so we once again start by writing a kernel (e.g. a matrix product), which we compile and disassemble. We then proceed as follows, in a systematic manner:

1) figure out the most probable semantics of the instruction using its name
2) run the program on the GPU and on the simulator, verify if the results are equal
3) if the results are not the same, execute the program step by step in the simulator and on the GPU with cuda-gdb
4) find the first instruction for which the state (register and memory values) in the simulated and executed program differ
5) collect results and source operands on both sides
6) find the purpose of this instruction in the executed program
7) implement this semantics in the simulator
8) try again from 2)

C. State of the implementation and current limitations

As of today, our implementation of PasTiS supports integer and floating point arithmetic instructions, memory accesses to all memories of the GPU, conditional branching and instructions for thread divergence/reconvergence handling, scheduling instructions, memory fences and synchronization barriers. Functionally speaking, the floating point arithmetic is handled natively by the machine that performs the simulation, so results may differ from the ones obtained on the Jetson TX2.

We currently simulate a direct-mapped cache hierarchy. In our experiments, this policy is enough to get precise results in terms of cache misses, but we expect this precision to fall as we experiment with ever larger kernels. As a consequence we are currently working on experimenting other policies, based on existing results [18].

[7]https://github.com/NervanaSystems/maxas

Another limitation is the warp scheduler whose policy remains unknown at this point. For now, a warp is simulated until its completion or until it reaches a synchronization barrier, at which point another warp is elected for simulation. Using this policy the simulation is functionally correct but does not faithfully represent the memory latency hiding mechanisms that the real GPU implements. As a consequence, our simulation timing results are close to the measurements on the actual GPU as long as at most one warp is active on each SMP, but the overhead is growing as we add more warps to the simulation. Determining a realistic warp scheduling policy is the next step in our research. To do so, we can rely on existing work [21] and on our simulator: once again we will work by trial and error, by first assuming a policy, implement it and validate it or not following the results of the simulations.

V. Performance evaluation

We conducted two kinds of experiments to assess the functional correctness and the timing precision of PasTiS. Since our objective is the simulation (and in a second step the static analysis) of neural network applications, we started by experimenting PasTis on integer matrix multiplications, which represent the major building block of neural network computations. We then implemented a simple feed-forward neural network for handwritten numbers recognition which we trained on the MNIST dataset [1] and tested our simulator on the inference function of this network.

The integer matrix multiplication benchmark uses the shared memory as is usually the case in "real-life" kernels: the program starts by a copy of the contents of the matrices to multiply from the global memory to the shared memory, then each thread performs the computation of one element of the result matrix and stores it directly in the global memory. We started the evaluation with a multiplication of two 4×4 integer matrices, which is handled by 16 threads of one warp (the 16 other threads are not active for this computation). The results are depicted in Figure 10. First of all the results are functionally correct: we obtain the same result matrix in the simulation as in the actual execution on the Jetson TX2. Second, the number of reported transactions to the shared memory is the same in the simulation and in the execution. Finally, the number of cycles reported in the simulation is 9.7% higher than in the measurements performed on the execution. We expected an overhead since we do not know all the implementation details of the GPU. In this experiment, we believe that the overhead is linked to the L2 cache of the GPU: in the Jetson TX2, the copy of the input data (the two input matrices) from the CPU to the GPU is performed through the L2 cache of the GPU. As a consequence, the cache is warm when the execution of the kernel starts on the GPU. Unfortunately, PasTiS does not simulate the copy of the inputs from the CPU to the GPU: it considers that the inputs are already in the global memory and thus starts the simulation of the kernel with a cold cache. We performed the same experiment on 8×8 (resp. 11×11 matrices), in order to test the simulator with 2 (resp. 4) active warps in parallel.

size : 4*4, block : 4*4 threads			
Measures	GPU	Simulator	Overhead
shared memory reads	8	8	0
shared memory writes	2	2	0
cycles	1131	1241	9.7%

Fig. 10. Matrix multiplication on one warp

size : 8*8, block : 8*8 threads			
Measures	GPU	Simulator	Overhead
shared memory reads	32	32	0
shared memory writes	4	4	0
cycles	1381	1491	7.9%

Fig. 11. Matrix multiplication on two warps

Since our target GPU is composed of 4 SMP each capable of running one warp in parallel, this is the maximum number of warps that can be executed/simulated without influence from the warp schedulers policy. Once again, the results of the simulation are functionally correct, and we obtain an overhead of 7.9% (resp. 6.1%) in terms of execution cycles (reported in Figures 11 and 12 respectively). We consider that these results validate our model of the shared memory.

Our neural network benchmark is a simple feed-forward network composed of 3 dense layers which performs the classification of input images representing handwritten digits (taken from the MNIST dataset). The first (resp. second, third) layer is composed of 784 (resp. 128, 64) weights and 128 (resp. 64, 10) biases, and the activation function is ReLU for all three layers. Each layer is implemented as a separate kernel which performs a matrix multiplication analogous to the one of our first experiment, and then calls an activation function. The network was written in C/CUDA, trained on the Jetson TX2, and the trained parameters were exported so they could be used for inference in the simulation as well.

In the experiments we first measured the difference between the simulated and executed floating point operations and its influence on the results of the network. As expected, we measured a slight difference in the computed probabilities (the result layer). However the difference only appears after the 14th decimal digit, and does not affect the classification results: the classification precision of the network is 98.4375% both on the Jetson TX2 and on PasTiS. Unfortunately, this benchmark heavily relies on the 64-bit floating point unit and on some hardware prefetch optimizations whose behaviour is not correctly modeled yet in PasTiS. As a result the cycles count of the simulation sees a more than 100% overhead compared to the actual measurement. We are currently working on correcting this issue.

VI. Related work

With the ongoing adoption of GPUs in embedded and time-critical systems, the real-time community has started to work on the characterization of the timing aspects of GPUs. Since NVIDIA is currently the leader in the market of GPGPUs, and

size : 11*11, block : 11*11 threads			
Measures	GPU	Simulator	Overhead
shared memory read	88	88	0
shared memory write	8	8	0
cycles	1580	1677	6.1%

Fig. 12. Matrix multiplication on four warps

in particular for the embedded systems segment, most of the research has focused on understanding and taming the NVIDIA GPUs. The main problems with these GPUs lie in their closed nature: NVIDIA does not provide a complete documentation of its GPUs, and many aspects that are relevant to their timing behavior is undocumented. As a result, most of the scientific production yet has been dedicated to reverse-engineering various aspects of the execution of kernels on NVIDIA GPUs with three major drawbacks. First, it is extremely tedious and time-consuming. Second, we approach the GPUs as black boxes and rely on experiments and measurements to characterize their behavior, with no guarantee that a particular case was not missed. Finally, as NVIDIA builds new GPU families, there is no guarantee that a new-generation GPU will still behave in the same way than the previous ones: the reverse engineering experiments must be conducted again to validate the hypothesis again. The same is true for the CUDA API and drivers, as well as for the instruction set architecture which are also closed source.

At the higher level of description, the community has so-far covered the behavior of the stream queues which control the interface between the CPUs and the GPU [3], [25], as well as the timing aspects of the memory copies between the CPU and the GPU memory spaces [7]. The memory hierarchy and scheduler hierarchy (at the block and warp mapping levels) has also been documented for particular GPU families [16], [18], [21], although no reverse engineering has yet been performed to unravel the exact warp scheduling policy.

A noticeable exception is [22], in which the authors weigh the pros and cons of using AMD GPUs for research in the real-time domain. The authors report that the general organization of AMD GPUs does not differ from the NVIDIA ones, with mainly small dimensioning differences (number of SMs, number of threads per warp), and in the terminology. One important difference is the absence of integrated GPUs (iGPUs: SoCs in which the GPU and CPUs share the same memory) in the AMD offer, and a more flexible choice in the mapping of the thread blocks in the AMD GPUs. According to the authors, the main point in favor of AMD is that they decided to make their software stack (in particular their drivers and APIs) open source and to vastly document it. However the authors report that the vast amount of documentation is not centralized, making it tedious to find information (the same as for NVIDIA GPUs, although for other reasons), and that the API and drivers code is not yet stable enough to make it completely worthwhile to invest time in understanding its inner workings and in designing compatible real-time modules

since they may not be compatible to the next revision of the API (the equivalent for NVIDIA GPUs to reverse engineering a particular aspect of a particular GPU which may behave differently in the next family of GPUs).

Other works are dedicated to defining WCET analysis methods for more abstract GPUs (usually a simulated GPU e.g. [2] with simplifying assumptions) [5], [6], [12]–[15]. The main limitation in these works is the simplifying assumptions made on the execution behavior of the GPUs: although the methods work for the target simulated GPU, they are either incomplete (focusing on a particular aspect) or cannot be applied to COTS GPUs which implement more complex hardware optimization strategies.

VII. Conclusion

This paper presents how we reversed-engineered some aspects of the NVIDIA Pascal GPU of the Jetson TX2 board, and how we implemented them in the PasTiS simulator that we are currently developing. PasTiS supports the actual NVIDIA SASS assembly language, which makes it capable of simulating kernels that are compiled for Pascal GPU targets. We first explained the hidden mechanisms responsible for handling thread divergence and reconvergence in the presence of conditional branch instructions. Then we uncovered how transactions to the shared memory are handled by the hardware. In PasTiS, we have modeled these mechanisms as well as a consequent part of the Maxwell/Pascal ISA. We were able to demonstrate the current accuracy of the simulator on benchmarks implementing matrix products with the use of shared memory. In our experiments, the simulator cycle count suffers is overestimated by less than 10% compared to the actual measurements on the board, and this overhead is reduced as the size of the matrices grows. This confirms that our model of the shared memory is correct.

As part of future work we intend to apply further our methodology to other architectural elements (in particular cache hierarchy, FP64 units and warp scheduler) in order to complete our simulator, until we are able to simulate neural networks with a high temporal precision. We will then release the simulator code as open source. Once the model is precise enough, we will use this knowledge to build static analysis models in order to derive the worst-case execution time of kernels running on the Jetson TX2 board.

References

[1] Mnist handwritten digits data set. https://deepai.org/dataset/mnist.

[2] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers. General-purpose graphics processor architectures. In *Synthesis lectures on computer architectures, Morgan & Claypool publishers*, 2018.

[3] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *8th International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, 2010.

[5] K. Berezovskyi. Timing analysis of general-purpose graphics processing units for real-time systems: Models and analyses. In *PhD dissertation, University of Porto*, 2015.

[6] K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for gpu threads running on a single streaming multiprocessor. In *2012 24th Euromicro Conference on Real-Time Systems*, 2012.

[7] A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia. Understanding and exploiting the internals of gpu resource allocation for critical systems. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[8] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr. Divergence analysis and optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[9] R. I. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1), May 2019.

[10] R. Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.

[11] A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang. Decoding cuda binary. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, 2019.

[12] V. Hirvisalo. On static timing analysis of gpu kernels. In *Workshop in WCET Analysis*, 2014.

[13] Y. Huangfu and W. Zhang. Static wcet analysis of gpus with predictable warp scheduling. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, 2017.

[14] Y. Huangfu and W. Zhang. Wcet analysis of the shared data cache in integrated cpu-gpu architectures. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[15] Y. Huangfu and W. Zhang. Wcet analysis of gpu l1 data caches. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018.

[16] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv*, apr 2018.

[17] Y.-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12), 1997.

[18] X. Mei and X. Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[19] Nvidia. Across thread out of order instruction dispatch in a multithreaded microprocessor. https://patentimages.storage.googleapis.com/ab/e4/d4/487e7e837f2ade/US7676657.pdf.

[20] Nvidia. Execution of divergent threads using a convergence barrier. https://patentimages.storage.googleapis.com/42/1d/77/18beae47a1fc64/US20160019066A1.pdf.

[21] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna. Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[22] N. Otterness and J. H. Anderson. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 2020.

[23] T. Ratsiambahotra, H. Casse, and P. Sainrat. A versatile generator of instruction set simulators and disassemblers. In *Int'l Symp. on Performance Evaluation of Computer Telecommunication Systems*, 2009.

[24] M. Schlickling and M. Pister. A Framework for Static Analysis of VHDL Code. In *7th International Workshop on Worst-Case Execution Time Analysis*, 2007.

[25] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, 2018.