Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective

Ignacio Sañudo Olmedo*, Nicola Capodieci*, Jorge Luis Martinez*†, Andrea Marongiu* and Marko Bertogna*
*University of Modena and Reggio Emilia, Modena, Italy
†Technical University of Munich, Munich, Germany
Email: name.surname@unimore.it

***Abstract*—Over the last few years, the ever-increasing use of Graphic Processing Units (GPUs) in safety-related domains has opened up many research problems in the real-time community. The closed and proprietary nature of the scheduling mechanisms deployed in NVIDIA GPUs, for instance, represents a major obstacle in deriving a proper schedulability analysis for latency-sensitive applications. Existing literature addresses these issues by either (i) providing simplified models for heterogeneous CPU-GPU systems and their associated scheduling policies, or (ii) providing insights about these arbitration mechanisms obtained through reverse engineering. In this paper, we take one step further by correcting and consolidating previously published assumptions about the hierarchical scheduling policies of NVIDIA GPUs and their proprietary CUDA application programming interface. We also discuss how such mechanisms evolved with recently released GPU micro-architectures, and how such changes influence the scheduling models to be exploited by real-time system engineers.**

I. Introduction

The sheer amount of data to be processed by next-generation embedded platforms is creating new challenges to real-time systems designers. In order to cope with massively parallel and data-hungry algorithms, the industry started to adopt hardware accelerators, e.g., programmable logic (FPGA), Graphic Processing Units (GPUs) and various types of Application-Specific Integrated Circuits (ASIC), like the increasingly widespread neural network inference accelerators.

The adoption of heterogeneous platforms featuring a multicore *host* coupled with different accelerators makes it more and more difficult to guarantee safety and real-time requirements as the complexity of the system-on-chip (SoC) increases. While commercial heterogeneous SoCs are very appealing – cost- and performance-wise – their very complex designs, often completely closed-source, prevent embedded systems engineers from gaining a detailed understanding of the temporal behavior of the various workloads that can coexist in the system, ultimately losing control of when and how outputs are produced.

Over the past few years, NVIDIA architectures have become a popular choice in many latency-sensitive domains, mostly thanks to a solid and simple programming model (CUDA) which allows also non-expert GPU developers to quickly achieve impressive performance-per-watt targets compared to traditional CPUs. The parallelism of GPU architectures is growing exponentially. Focusing on NVIDIA, the integrated GPU of the Jetson TX2 platform features 256 SIMD cores, while the core count of the discrete GPU featured in the Pegasus AGX platform adopted in many autonomous driving prototypes has grown up to 2816 computing units.

Historically, the GPU hardware and software development kits have been designed for very high peak and average performance (throughput), completely neglecting timing predictability. The typical execution pattern for a heterogeneous CPU-GPU system is the following [1]: 1) move data from the *host* CPU to the GPU device; 2) execute the compute kernel (i.e., the part of the program containing abundant parallelism) on the GPU device; 3) move data from the device back to the *host*. This submission paradigm can be modelled as a non-preemptive FIFO, in which different applications can access the GPU in a mutually exclusive manner. In modern GPUs, such a naive *offloading* strategy may lead to severe GPU under-utilization.

To overcome this effect, CUDA introduced a software mechanism called *CUDA streams*. A *stream* in CUDA is defined as a workload queue for the device, where kernel execution requests can be enqueued (offloaded) by the *host* asynchronously, in a non-blocking fashion. The GPU device then dispatches work requests from *streams* onto the GPU compute clusters based on the current *occupancy*, i.e., the utilization of processing and memory resources local to the accelerator clusters.

Moving to a model in which multiple GPU kernels can time-share GPU resources improves performance, but complicates the design of real-time, safety-critical software [2]. Unfortunately, the details of how the NVIDIA GPU hardware scheduler subdivides streams of work among the compute clusters (*streaming multiprocessors* or *SM* in CUDA terminology) are not publicly available. The lack of information about the internals of NVIDIA hardware and software blocks complicates the development of reliable models for system timing analysis. Several such models proposed in the literature either treat the GPU as a black box, or assume a simplified structure that unfaithfully represents the real system. For example, the default block-to-SM distribution policy in the GPU is round-robin (RR), but the scheduling algorithm gets much more complex when multiple streams are used. Simplistically assuming RR in this case [3], [4] can lead to severe misprediction of the block execution times. Figure 3 shows the misprediction rate of a model of a block-to-SM scheduler based on RR, plotted as a function of the number of

2642-7346/20/$31.00 ©2020 IEEE
DOI 10.1109/RTAS48715.2020.00022

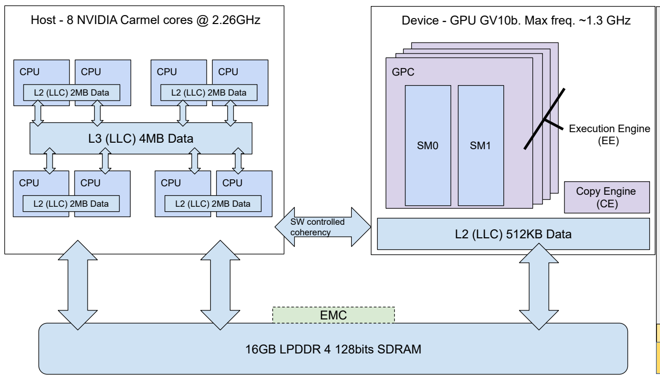


Figure 1: Block diagram of the NVIDIA Xavier SoC.

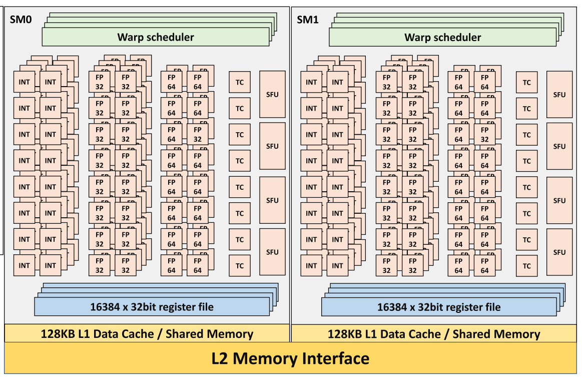

SM0
SM1
Warp scheduler
INT
FP 32
FP 64
TC
SFU
16384 x 32bit register file
128KB L1 Data Cache / Shared Memory
L2 Memory Interface

Figure 2: GPC architecture in the Volta iGPU.

streams per kernel. The values displayed in the bars are derived from 7000 launch configurations of a target *kernel*[1]. For each configuration we compare the RR block-to-SM distribution with that observed on an NVIDIA Xavier platform (see Section II). If they differ, we consider it a misprediction. The figure shows that as the number of streams increases the probability of misprediction gets as high as 96%. Understanding the scheduling decisions of modern GPUs is a necessary step to allow real-time engineers to safely model the performance of these accelerators, as failing to do so can significantly impact the predictability of concurrent CUDA streams. Yet, to this day there is no comprehensive description of how the scheduler distributes the work among the available SMs, nor how such mechanisms changed in recently released NVIDIA GPU micro-architectures. In this paper, we provide an in-depth analysis of how the GPU hardware scheduler works, starting from the CUDA stream dispatcher, to the actual scheduling of individual *warps*, the minimal scheduling unit of each SM.

This paper is organized as follows: In Section II we describe the most prominent architectural features of an NVIDIA GPU and the key programming model constructs offered by CUDA to exploit such features. Section III provides an in-depth overview of the NVIDIA scheduler hierarchy, with Sections IV and V dissecting the available mechanisms inside and outside each SM, respectively. Each of these sections provides detailed discussion on performance and predictability aspects, along with extensive experimental evaluation. In Section VI we provide an overview of related work on understanding the scheduling and performance details of NVIDIA GPUs, summarizing the novelties of our contribution. Section VII concludes the paper, highlighting future research directions.

[1]Using [2-8] streams, [1-4] blocks per kernel, [1-1024] threads per block.

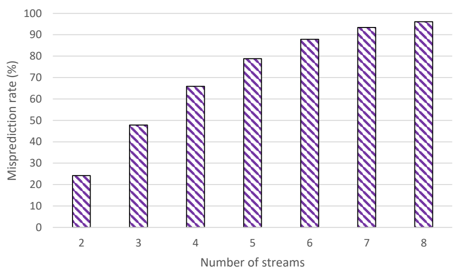


Figure 3: Misprediction rate of a round-robin block-to-SM mapping model compared to the real mapping.

II. Background on NVIDIA GPUs

A. GPU hardware

Modern heterogeneous SoCs (hSoC) are typically composed of a multi-core CPU host, one or more hardware accelerators (devices) and the related memory interface and hierarchy. Figure 1 depicts a schematic representation of the most recent NVIDIA Tegra hSoC, codenamed *Xavier*. The hSoC incorporates an NVIDIA-proprietary design featuring an eight-core *host* processor compliant with the ARM v8.2 architecture (*Carmel*), and an integrated GPU (iGPU) based on the *Volta* microarchitecture. The system RAM consists of a 16-GB LPDDR4x module, shared among all the computing devices.

Internally, the iGPU features 512 cores organized in computing clusters called *Streaming Multiprocessors* (SMs). More SMs can be grouped within larger clusters named *Graphic Processing Clusters* (GPCs), sharing a common interface towards the 512KB L2 memory (a unified cache). Multiple GPCs also share a DMA engine, called *Copy Engine* or *CE* in CUDA terminology. The computing resources within all GPCs also are called *Execution Engines* or *EE* in CUDA terminology.

The architectural details of a GPC in the Xavier platform are depicted in Figure 2. Here a GPC is composed of two SMs sharing the physical interface towards the L2 memory. Internally, each SM is partitioned into four processing blocks, each with 16 INT32 cores, 16 FP32 cores, 8 FP64 cores, 2 mixed-precision Tensor Cores (TC) for deep-learning matrix arithmetic, one special function unit (SFU) one warp scheduler and a 64KB register file. Globally, all the processing blocks share a L1 data cache plus a L1 shared memory (a fast scratchpad memory), with a cumulative capacity of 128 KB.

B. CUDA Execution and Programming Model Basics

In its most simple incarnation, the CUDA execution model envisions three steps: (i) the host CPU copies data and program

Listing 1: Naive host CUDA code

```
1  void cpuThread(){
2   // data init and allocation
3   const size_t msize =
        ↪ W_MATRIX*H_MATRIX*sizeof(float);
4   float *h_iData = (float*)malloc(msize);
5   float *h_oData = (float*)malloc(msize);
6   float *d_iData;cudaMalloc(&d_iData,msize);
7   float *d_oData;cudaMalloc(&d_oData,msize);
8
9   fillInputData(h_iData, ...);
10  dim3 dimGrid(W_MATRIX/TILE_DIM,
        ↪ H_MATRIX/TILE_DIM, 1);
11  dim3 dimBlock(TILE_DIM, BLOCK_ROWS, 1);
12
13  // actual command submission
14  cudaMemcpy (d_iData, h_iData, msize,
        ↪ cudaMemcpyHostToDevice);
15  transpose<<<dimGrid, dimBlock>>>
        ↪ (d_Odata,d_iData);
16  cudaMemcpy (h_oData, d_oData, msize,
        ↪ cudaMemcpyDeviceToHost);
17  cudaDeviceSynchronize();
18  // data post-processing
19 }
```

code from its memory domain to memory accessible by the GPU; (ii) the GPU executes the offloaded *kernels*; (iii) data produced by the offloaded computation is copied back to the host memory. These operations are abstracted as *copy* and *execution* commands that can be enqueued in the GPU scheduling hardware, to be performed by the (*CE*) and (*EE*), respectively. Once data is available to the GPU, the CUDA programming model follows a Single Instruction Multiple Threads (SIMT) paradigm, where the same instructions are issued to multiple threads working on different data items. Threads are logically organized in arrays (groups) of 32, called *warps*. A *warp* is the minimum schedulable entity in the GPU.

Besides the SIMT parallelism within a *warp*, the GPU can leverage other dimensions of parallelism (multiple warps within an SM, multiple SMs in the GPC and in the whole device). To enable the exploitation of this abundant parallelism CUDA kernels are launched as *grids* of *blocks* of *threads*. A thread *block*, also called a *Cooperative Thread Array* (CTA), is an abstraction used to specify a group of *warps* that can be executed in parallel. The *warp schedulers* within each SM are in charge of distributing the threads to the available hardware resources. Thread blocks can be organized as 1D, 2D and 3D grids. The hardware scheduler treats each block as a multiple of 32 threads (i.e., a *warp*) regardless of the effective number of threads defined in the kernel invocation.

Listings 1 and 2 show the simplified CUDA code [5] to implement *host* (CPU) and *device* (GPU) operations to transpose a W_MATRIX x H_MATRIX matrix of `float` elements. Focusing on the *host* code, the first step is to allocate memory for *host* and *device* data (lines 3-7): for *host* data, a regular

Listing 2: Naive device CUDA code

```
1  __global__ void transpose(float *odata,
       ↪ const float *idata){
2   int x = blockIdx.x*TILE_DIM+threadIdx.x;
3   int y = blockIdx.y*TILE_DIM+threadIdx.y;
4   int width = gridDim.x*TILE_DIM;
5
6   for (int j=0;j<TILE_DIM;j+=BLOCK_ROWS)
7     odata[x*width + (y+j)] =
          ↪ idata[(y+j)*width + x];
8  }
```

`malloc` is used, whereas the CUDA API runtime function `cudaMalloc` is used for *device* data. *Host* memory is then initialized (line 9) before the kernel launch configuration is defined (lines 10, 11). The `dim3` data structure hosts three integer values that represent the number of *threads* or *blocks* along the x, y and z dimensions. By defining the organization of the *compute grid* along each dimension the programmer implements a *tiling* scheme for the offloaded work. A *tile* is a portion of the input matrix: each *block* contains enough threads to operate on that portion of the matrix. In this example, thread *blocks* are defined in two dimensions only (as the matrix is bidimensional), with the third dimension set to 1. `dimGrid` contains the number of blocks for the next kernel invocation ($matrix\_width/height/TILE\_DIM$), while `dimBlock` defines the number of threads per block ($TILE\_DIM * BLOCK\_ROWS$).

Copy commands are issued to transfer data from the *host* to the *device* (line 14) and back (line 16). The actual kernel offloading happens at line 15. The name of the kernel (`transpose`, in this example) must match the entry-point of the *device* code, shown in Listing 2.

Focusing on the *device* code, the first thing to notice is that work partitioning happens via thread indexing (lines 2 - 4): thread IDs range from 0 to the number of threads in a block - 1. Retrieving the thread ID within the block along the x and y dimension relies on the CUDA keyword `threadIdx.x/y`. Similarly, the block ID can be retrieved via the CUDA keyword `blockId.x/y`. The number of blocks along the x dimension can be queried via the `gridDim.x` keyword.

Based on these IDs, each worker thread will compute the offset at which to index its own share of the input/output data. In this example `TILE_DIM=32`, `BLOCK_ROWS=8` and both `iData` and `oData` are 1024x1024-element matrices. The loop at line 6 tells us that each thread in the block reads 4 elements from `iData`, with a stride of 32*8 elements and copies them with reversed `x`, `y` coordinates in `oData`.

C. Advanced CUDA Execution Model and Programming Constructs

CUDA allows to lift the limitations of its basic execution model by providing constructs to define the execution of interleaved compute and data operations. This is achieved with

Listing 3: Optimized host CUDA code

```
1  void cpuThread(){
2   const size_t msize =
        ↪ W_MATRIX*H_MATRIX*sizeof(float);
3   float *h_iData;
        ↪ cudaMallocHost(&h_iData,msize);
4   float *h_oData;
        ↪ cudaMallocHost(&h_oData,msize);
5   float *d_iData;cudaMalloc(&d_iData,msize);
6   float *d_oData;cudaMalloc(&d_oData,msize);
7
8   cudaStream_t s; cudaStreamCreate(&s);
9   fillInputData(h_iData, ...);
10  dim3 dimGrid (W_MATRIX/TILE_DIM,
        ↪ H_MATRIX/TILE_DIM, 1);
11  dim3 dimBlock (TILE_DIM, BLOCK_ROWS, 1);
12
13  cudaMemcpyAsync(d_iData, h_iData, msize,
        ↪ cudaMemcpyHostToDevice, s);
14  transpose<<<dimGrid, dimBlock, 0, s>>>
        ↪ (d_Odata, d_iData);
15  cudaMemcpyAsync(h_oData, d_oData, msize,
        ↪ cudaMemcpyDeviceToHost, s);
16  // cpu work might be performed here,
17  // while the gpu is working
18  cudaStreamSynchronize(s);
19 }
```

CUDA streams. A CUDA *stream* is an abstraction of a queue of commands offloaded to the GPU; a software entity that has to be created and specified for each copy and compute command. If a *stream* is not specified, the CUDA runtime uses the default *stream* (*Stream* 0). Its standard behaviour implies implicit synchronization with every other *stream* from the same application. Commands pushed in different *streams* can be interleaved and, whenever possible, they run concurrently. Commands in the same *stream* are ordered in a FIFO fashion and cannot overlap. *Stream* execution can be either synchronous or asynchronous. If synchronous, the CPU thread dispatches the work to the GPU and blocks until the kernel completes. If asynchronous, host-side computation is possible during GPU execution. In both cases, upon GPU kernel completion the CE copies back the data computed by the accelerator, if the CPU or other compute devices need it.

The use of CUDA *streams* implies the knowledge of a few more advanced constructs: *pinned host memory* and *device shared memory*. To illustrate these concepts, Listing 3 shows an optimized version of the matrix transpose *host* code. CPU data allocation is now performed through the `cudaMallocHost` function (lines 3, 4). While a regular `malloc` allows the developer to allocate *pageable* memory, `cudaMallocHost` requests *pinned* memory, i.e. memory that is allocated in an area in which paging is bypassed. This drastically improves the bandwidth for data transfers. *Pinned* memory also enables overlapping of copy and compute operations within different CUDA streams.

A CUDA *stream* is created at line 8: this is the queue of commands where copy and compute operations are inserted. Data transfers from/to the *host* are enqueued through the `cudaMemcpyAsync` function (lines 13, 15), which – different from the `cudaMemcpy` – allows the CPU to perform other work while copies are pending.

The kernel launch syntax becomes slightly more complicated when using streams (line 14): besides *grid* and *block* dimensions, in the triple bracket structure we now have to indicate the amount of *dynamic shared memory* used by the kernel (0 Bytes in the example) and the stream in which we want to enqueue the kernel command. The device *shared memory* is a fast and local per SM scratchpad memory, accessible by all threads within a block. Accessing it is considerably faster than accessing memory devices located outside the SM (GPU last-level cache, system DRAM). Requesting *shared memory* dynamically via the triple bracket syntax on the *host* side, as we just discussed, is useful whenever the size of the data to be offloaded is not known at compile time. When the size of the data is known at compile time it can be statically allocated in *shared memory* by using the `__shared__` variable declaration specifier in the *device* code.

Note that commands are all pushed to the same stream `s`, hence they will be executed on the GPU in that same order. If by the time the CPU reaches the `cudaStreamSynchronize` function call at line 18 the GPU is still consuming the commands pushed to the stream(s), then the CPU thread will block.

When using *shared memory* in the *device* code data must be explicitly copied from/to the main memory. For performance, memory accesses to the main memory (i.e. outside the SM) should always be *coalesced*. Coalesced main memory accesses greatly improve the use of memory bandwidth, and are achieved when parallel threads within the same *warp* access contiguous memory locations; this enables combining different memory access requests into a reduced number of coalesced requests hitting the same cache lines.

III. GPU Scheduling hierarchy

GPU scheduling in NVIDIA architectures can be seen as a hierarchical arbitration mechanism. As illustrated in Figure 4, at the top level of the scheduling hierarchy lies the *application scheduler*. A CUDA application might use multiple *streams*, the scheduling of which involves different levels of the hierarchy. *Kernels* launched in different streams are composed of *blocks* of *threads*. More specifically, *streams* are pushed in a FIFO queue. The thread *blocks* are then distributed among the SM cores with a block scheduler. Finally, within the *blocks* dispatched to the same SM, the last level of the scheduling hierarchy manages the *warps* composing the currently active *CTAs* (or blocks).

A. NVIDIA Application scheduler

A detailed description of the scheduling policy at the highest level of the GPU arbitration mechanism has been presented in [6] and [7]. At the driver-level, each application that needs the GPU opens a number of *channels*, which are inserted in a *runlist*. Each entry in the runlist is characterized by

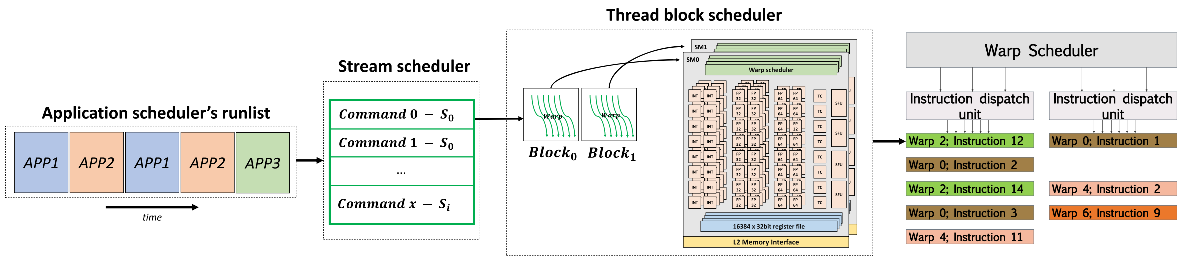


Figure 4: GPU scheduling hierarchy. Streams from S_0 to S_i belong to an application selected from the *runlist*.

a *timeslice length* and a *priority* value (low, medium and high). This whole mechanism is therefore described as a work-conserving TDMA (Time Division Multiple Access) among channels, where each channel can be assigned multiple slots within the runlist, according to its priority value, thus shaping the sequence of slots for the TDMA round. Preemption occurs at the end of the timeslice of the currently running channel at different granularities: CUDA thread block or CTA boundary, CUDA Kernel instruction boundary, or at pixel-level in purely graphic contexts. We refer the interested reader to the cited contributions for more in-depth discussion on the NVIDIA interleaved scheduler and related response time analysis. However, one detail is worth further investigation: the cited contributions and the NVIDIA Drive [2] documentation hint that only the channels belonging to a single application are considered by the scheduler at a given time. In section V-A, we will show that this is not always the case.

B. Stream scheduler

The stream scheduler is the entity in charge of arbitrating work submitted to different streams of an application. Stream operations are introduced in FIFO order according to their submission time. It is important to highlight that CUDA streams relate to a single application. For example, let us suppose the following scenario: Application A_0 has 2 streams and Application A_1 has 1 stream. If application A_0 is currently running, streams of application A_1 are not supposed to interfere with A_0, as this is a direct consequence of the isolation between contexts imposed by the NVIDIA application scheduler. If A_0 is running, then the stream scheduler has to take a decision with respect to the streams utilized by A_0. Operations within the same stream are ordered in a FIFO fashion and execute sequentially, whereas operations in different streams are unordered and can execute in parallel. Starting from the Maxwell GPU micro-architecture (featured in the Jetson TX1 embedded board) and subsequent micro-architectures, CUDA provides a runtime function call for assigning priorities to streams. At the time of writing, all the tested GPU micro-architectures (Maxwell, Pascal, Volta and Turing) feature only two discrete levels of priorities (high and low). If a low priority stream is currently occupying all the compute resources of an SM, a kernel submitted later in time on a high-priority stream can preempt the currently running kernel. In this case, preemption occurs at CTA boundary [8]. Further details about priority assignments of CUDA streams can be found in [9].

C. Thread Block Scheduler: Block-to-SM mapping

In NVIDIA GPUs, SM identifiers range from 0 to the maximum number of available SMs minus one. If all kernels are dispatched to a single stream, thread blocks will be distributed through all the available SMs in a RR fashion, beginning with even-ID SMs and then proceeding with odd-ID SMs in increasing ID order [10]. The thread block scheduler – also called CUDA work distributor (CWD) in old architectures – performs an *occupancy* test before assigning a block to an SM, in such test, the status of each SM is inspected to determine its current degree of resource utilization [3]. The test aims at assessing whether the current occupancy is such that a new block can be allowed into the target SM (i.e., if the unused compute and memory resources are sufficient to satisfy the new kernel's demand), with the final goal of mapping thread blocks to SMs.

The factors that dictate the occupancy level of a kernel are: (i) number of threads/warps per thread block; (ii) shared memory per thread block; (iii) number of registers per warp. An SM is considered fully utilized when all the warp schedulers have some instruction to issue for some warp at every clock cycle. When this happens, the latency of memory operations is hidden. The CUDA Occupancy Calculator[3] is a publicly-available spreadsheet distributed by NVIDIA that helps computing the theoretical occupancy of a target GPU given a specific thread/block configuration. Combining the use of this calculator with the architectural parameters obtained via the *deviceQuery* command[4], we derive Equation 1. This formula can be used to obtain the percent utilization of *thread*, *shared memory* and *register* resources.

$$Occupancy = \ max \left(\left\lfloor \frac{MaxThreads}{NWarp * 32}, \frac{T_{SHM}}{(U_{SHM} + \epsilon) * NB}, \frac{dim_{RegFile}}{Regs * NWarps} \right\rfloor \right) where\ NWarp = \left\lceil \frac{\#Threads}{32} \right\rceil \tag{1}$$

$MaxThreads$ is the maximum number of threads per SM; T_{SHM} represents the maximum amount of shared memory that

[2]https://www.nvidia.com/en-us/self-driving-cars/drive-platform/

[3]https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html

[4]A utility included in the CUDA SDK.

can be used for a single block; and $dim_{RegFile}$ denotes the size of the register file. These are all architecture-specific features. We summarize these values in Table I, for all the tested GPUs. $NWarp$ indicates the number of warps already allocated to the current SM, plus those requested by the new block, i.e., the ones that the scheduler is trying to accommodate on the SM, and it can be derived from the total number of threads dispatched by the kernel ($\#Threads$). U_{SHM} is the shared memory allocated by the kernel already in execution on the SM, plus that requested from the new kernel block. ϵ represents a minimum threshold of 3KB of allocated shared memory, below which the scheduler disregards shared memory usage when deciding where to allocate the new kernel to. More details are provided in Section III-C2. NB is the number of blocks already launched. Similarly, $Regs$ is the number of registers used by the kernel already allocated to the SM, plus those requested by the new kernel.

Normally, an expert programmer tries to maximize the occupation of the GPU, with the aim of improving the performance. Distributing the workload across the multi-dimensional thread grid (i.e., determining the number of threads/warps per block) is a means to maximize the SM occupancy value, just like sizing the data structures so as to use as much *shared memory* as possible. Playing with register usage is also controllable to some extent, as the programmer can limit the number of registers available for each warp during compilation. However, while in general low occupancy implies low instruction issue efficiency and thus sub-optimal kernel performance, maximizing occupancy does not necessarily lead to minimizing kernel execution times [11], as typically one of the three resources is saturated much earlier than the others[5]. The general assumption made in all previous work is that thread blocks are scheduled with a round-robin policy, provided that the maximum occupancy of an SM – as given in Equation 1 – is not exceeded. In the following sections, we show that this assumption is not always correct when multiple streams are used. In particular, we analyse separately the cases where each of the three terms in Equation 1 dominates (i.e., each of the three resource types is saturated), and derive models to describe the cases where block-to-SM mapping does not comply with the RR policy.

1) ***Block Scheduling when Thread Usage is Maximum****:* In contrast to the default round-robin policy, some specific configurations of blocks belonging to kernels coming from different streams cause the block scheduler to counter-intuitively allocate more than one block to the same SM, even if other SMs are idle.

We first focus on thread usage, neglecting register and shared-memory resource constraints in the kernel configurations of the submitted streams. In other words, the second and third terms of Equation 1 are dominated by the first term.

Unless otherwise stated all the following experiments are conducted on top of the Xavier platform (Volta GPU architecture), assigning an identical priority value to all involved CUDA streams. The maximum thread count for a single block is 2048 (64 warps) per SM. We assume all SMs to be initially idle. Two streams are launched, each stream having a single kernel in its queue. Each kernel is composed of one block. In order to determine how blocks are mapped onto SMs, we vary the number of threads (i.e., warps) per kernel and report the results of all the possible permutations in Table II. Varying the number of warps per kernel allows us to analyze the rationale behind the heuristic used to perform the block-to-SM mapping. This can be achieved by simply varying the number of threads within a block (see grid creation in Line 11 of Listing 3). To retrieve in which SM each block is scheduled, a thread per block performs a read from a special register. Specifically in device code, a thread with local ID equal to 0 executes the following assembly instruction $asm\ volatile("mov.u32 \%0, \%\%smid;" : "=r"(smid));$ This instruction returns the identifier of the SM on which a particular thread is executing.

Note that the same table correctly captures the behaviour of older architectures, as identical results were obtained in Maxwell- and Pascal-based chips. The first two columns and rows of the table show the number of threads and warps used by the first stream (rows) and the second stream (columns), respectively. Each other cell indicates the number of warps that an SM can still accommodate after the warps from the first stream (shown in the rows) have been assigned. We use color coding to indicate the cases when the scheduler chooses a different mapping than the default round-robin policy. More specifically, white cells show the cases when the two blocks are distributed to different SMs (the default policy), whereas light gray cells show the cases when both blocks are mapped onto the same SM. For instance, considering a launch configuration where the first stream uses 32 threads (1 warp) and the second stream uses 96 threads (3 warps), Table II reveals that both blocks are allocated onto the same SM, overriding the default round-robin policy. From the analysis of the whole table we can infer the following scheduling rule:

Definition III.1. *Let mw be the maximum number of schedulable warps per SM. Given a block with x warps launched by a stream and allocated to a given SM, a second block composed of y warps and launched by a different stream gets allocated to the same SM, if the following condition holds:*

$$mw - x \geq \left(\left\lfloor \frac{mw - y}{y} \right\rfloor + 1 \right) y \tag{2}$$

It can be observed that light gray cells appear only above the numbers in the diagonal (underlined in the table). These numbers have the form $mw - x$, or rather $mw - y$, as in the diagonal $x = y$. Furthermore it can also be seen that these cells correspond to the cases when the number of remaining warps, $mw - x$, is greater than or equal to the smallest number that is multiple of y and lies above the diagonal, i.e. both blocks

[5]And, in particular, typically programmers mostly focus on maximizing thread usage.

Table I: Different NVIDIA architectures and main related features.

Jetson TX2 (Pascal)	
# Multiprocessors	2
# CUDA Cores per SM	128
Maximum number of schedulable threads per SM	2048
# registers per block	65536
Total SHM per block	48KB
L2 cache size	512KB

Jetson Xavier (Volta)	
# Multiprocessors	8
# CUDA Cores per SM	64
Maximum number of schedulable threads per SM	2048
# registers per block	65536
Total SHM per block	48KB
L2 cache size	512KB

Pegasus AGX (Discrete Turing)	
# Multiprocessors	44
# CUDA Cores per SM	64
Maximum number of schedulable threads per SM	1024
# registers per block	65536
Total SHM per block	48KB
L2 cache size	4MB

Table II: Block-to-SM mapping common to the Maxwell, Pascal and Volta architecture.

#Threads		**32**	**64**	**96**	**128**	**160**	**192**	**224**	**256**	**288**	**320**	**352**	**384**	**416**	**448**	**480**	**512**	**544**	**576**	**608**	**640**	**672**	**704**	**736**	**768**	**800**	**832**	**864**	**896**	**928**	**960**	**992**	**1024**
	#Warps	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**
32	**1**	63	63	**63**	63	63	63	**63**	63	**63**	63	63	63	63	63	63	63	63	63	63	63	**63**	63	63	63	63	63	63	63	63	63	63	63
64	**2**	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	**62**	62
96	**3**	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61
128	**4**	60	60	60	60	**60**	**60**	60	60	60	**60**	60	**60**	60	60	**60**	60	60	60	60	**60**	60	60	60	60	60	60	60	60	60	**60**	60	60
160	**5**	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59
192	**6**	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	**58**	58	58	58
224	**7**	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	**57**	57	57	57	57	57	57	57	57	57	57	57	57	57
256	**8**	56	56	56	56	56	56	56	56	56	56	56	56	56	**56**	56	56	56	56	56	56	56	56	56	56	56	56	56	**56**	56	56	56	56
288	**9**	55	55	55	55	55	55	55	55	55	55	**55**	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55
320	**10**	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	**54**	54	54	54	54	54	54	54	54	**54**	54	54	54	54	54
352	**11**	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53
384	**12**	52	52	52	52	52	52	52	52	52	52	52	52	**52**	52	52	52	52	52	52	52	52	52	52	52	52	**52**	52	52	52	52	52	52
416	**13**	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	**51**	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51
448	**14**	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	**50**	50	50	50	50	50	50	50
480	**15**	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49
512	**16**	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	**48**	48	48	48	48	48	48	48	48
544	**17**	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47
576	**18**	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	**46**	46	46	46	46	46	46	46	46	46
608	**19**	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45
640	**20**	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	**44**	44	44	44	44	44	44	44	44	44	44
672	**21**	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43
704	**22**	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42
736	**23**	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
768	**24**	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
800	**25**	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39
832	**26**	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38
864	**27**	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
896	**28**	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36
928	**29**	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
960	**30**	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34
992	**31**	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
1024	**32**	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32

are allocated to the same SM if and only if $mw - x \geq k \cdot y$, where $k = \lfloor (mw - y)/y \rfloor + 1$. These observations provide the rationale for Equation 2.

Performing the exact same experiment on the Drive AGX Pegasus platform, which includes two discrete Turing GPUs and whose maximum number of threads per SM is 1024, results in Table III. By examining this new table, we can verify that our block allocation rule applies if we consider $mw = 32$. Moreover, further experiments (not included due to space constraints), where the maximum number of schedulable warps varies, show that we can extend the previous scheduling rule in the following manner:

Definition III.2. *Given a block with x warps launched by a stream and allocated to a given SM, whose total number of allocated warps was z prior to the allocation, a second block composed of y warps and launched by a different stream gets allocated to the same SM, if the following condition holds:*

$$(mw - z) - x \geq \left(\left\lfloor \frac{(mw - z) - y}{y} \right\rfloor + 1 \right) y \qquad (3)$$

This indicates that from a warp occupancy point of view, a Turing SM with 0 scheduled warps ($mw = 32, z = 0$) behaves in the same way as a Maxwell/Pascal/Volta SM with 32 warps already scheduled ($mw = 64, z = 32$).

This scheduling rule can be applied to kernels from different streams even if they are dispatched using multiple blocks per invocation. To explain this concept, a visual representation of two different scenarios performing a basic *Mandelbrot* computation dispatched in two streams is shown in Figure 5. To perform this experiment we used the CUDA scheduling mirror[6]. Specifically, in this experiment we consider two streams, S_0 and S_1, each featuring one kernel composed of 4 blocks. In a first experiment we consider that S_0 and S_1 are launched with 128 threads (4 warps) each. This is illustrated on the left side of Figure 5. As per the default mapping policy (i.e., round-robin), all blocks of S_0 and S_1 are fairly distributed across the SMs because all the blocks are dispatched with the same number of threads. In a second experiment (results shown on the right side of Figure 5) S_0 is launched with 128 threads (4 warps) and S_1 with 160 threads (5 warps) for each block. In this case it is observable that the mapping process performed does not follow a round-robin policy.

Let us consider the latter case in more detail. Following the described rules, the first 4 blocks of S_0 will be mapped onto SMs 0, 2, 4 and 6. To analyze where the blocks of S_1 are mapped, we evaluate Equation 2 for each SM. S_0 launches 4 warps and S_1 launches 5 warps, so $x = 4$ and $y = 5$. When S_1 is dispatched, the first condition that the block scheduler evaluates is if there are enough available resources to schedule the first block of S_1. This condition is satisfied since there are

[6]https://github.com/yalue/cuda_scheduling_examiner_mirror

Table III: Block-to-SM mapping of the Turing architecture.

#Warps	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
2	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
3	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
4	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28
5	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
6	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
7	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25
8	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
9	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
10	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
11	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
12	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
13	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
14	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
15	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
⋮																																
31	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

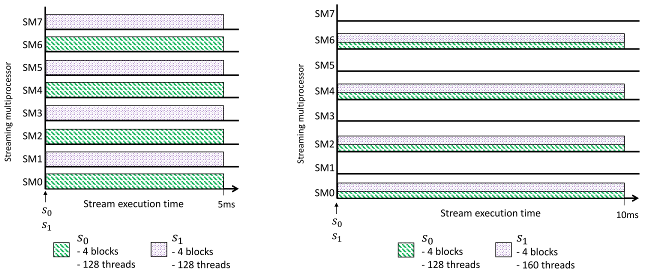


Figure 5: Streams S_0 and S_1 executing 2 instances of the Mandelbrot kernel. Block-to-SM mapping and related execution time is shown in case of (left) expected round-robin and (right) anomalous block-to-SM mapping.

4 SMs executing 4 warps and the rest of SMs are empty. Then the condition from Equation 2 is evaluated first against the current occupancy of SM 0. Thus,

$$64 - 4 \geq \left(\left\lfloor \frac{(64-0)-5}{5} \right\rfloor + 1 \right) 5$$

Hence, $60 \geq 60$. Equation 2 holds, therefore the first block in S_1 will be mapped onto SM 0. Then, the scheduler will consider the second block of S_1, and will check it against the same condition on the same SM 0, but this time with a different outcome. Hence, the same condition will be tested against the actual occupancy of SM 2 (recall that SMs with even IDs will be considered before the whole set of SMs with odd IDs). This will cause the second block of S_1 to be scheduled in SM 2. Eventually, all the blocks in S_1 will be scheduled onto the same SMs used by stream S_0, leaving half of the SMs idle. This may cause severe under-utilization issues of GPU resources, potentially harming performance and leading to larger worst-case response time bounds for both kernels.

Load Balancing: There are some scenarios, where the behavior of the thread block scheduler that we discussed so far does not hold, as the thread block scheduler aims at equally distributing the workload through the SMs. For instance, let us consider a kernel in a stream that launches 7 blocks, each containing 192 threads (6 warps). Assuming the GPU was initially fully idle, following the default scheduling rule, each block gets assigned to a different SM, leaving only one of the SMs idle. Let us now assume that two kernels from two streams, each kernel composed of only one block, are launched. Depending on the number of warps of the first kernel, its only block is assigned to either one of the non-idle SMs or to the empty one, according to Equation 3.

Table IV reports the allocation of the second kernel with respect to the first one, when the number of threads/warps composing the kernels vary. The first column and row represent the number of warps used by the first and second stream, respectively. Akin to the previous tables, numbers in the other cells indicate the number of remaining warps, after the first block is assigned to an SM. White cells denote the cases when the two blocks are mapped onto different SMs, light and dark gray cells denote scenarios where blocks are assigned to the same SM. Equation 3 predicts the block distribution encoded by white and light gray cells, but it fails to anticipate the block-to-SM allocation given in the dark-gray ones.

To better understand how the load balancing mechanism works we present the following experiment. In this experiment we assume that 6 SMs are occupied with one block of 512 threads (16 warps), then, the objective is to schedule 16 blocks of 64 threads (2 warps). The fact that all 16 blocks are assigned to the initially unoccupied SMs suggests that the scheduler aims at balancing the load of all the SMs. However, if the goal is to schedule 16 blocks of 128 threads (4 warps) instead, while the first 8 blocks are mapped onto the initially empty SMs, the remaining blocks are distributed among all the SMs. In other words, when all the SMs reach a balanced state, the distribution of the blocks turns to be round-robin until the number of warps per block change.

By repeating the experiment for a large number of scenarios where the work among the SMs is not evenly distributed, we can infer an extra allocation rule:

Definition III.3. *If initially each of the available SMs, except for one, is occupied by a block composed of z' warps and launched by the same stream, given a block with x warps launched by a second stream and allocated to a specific SM, a new block composed of y warps and launched by a third stream is to be allocated to the same SM, if the following condition holds:*

$$mw - z' < \left\lfloor \frac{mw - x}{y} \right\rfloor y \leq mw - x \tag{4}$$

Akin to the derivation of Equation 2, it can be noticed that light gray cells show up, when (i) the number of warps left after the allocation of the block with x warps, $mw - x$, is greater than or equal to the largest number that is multiple of y and is less than or equal to $mw - x$, i.e. $mw - x \geq k' \cdot y$, where $k' = \lfloor (mw - x)/y \rfloor$, and (ii) this number, $\lfloor (mw - x)/y \rfloor \cdot y$, is greater than the number of warps left in the SM where a block with z warps was allocated, i.e. $\lfloor (mw - x)/y \rfloor \cdot y > mw - z'$. Combining both conditions yields Equation 4.

In conclusion, our understanding is that the scheduler first attempts to find an allocation by means of Equation 3, resorting to Equation 4 if the first condition is not satisfied. This fact sheds light on the rationale behind the scheduler operation. Maximizing the utilization of the SMs appears to be the highest priority. After that, decisions are made so that the number of warps across all SMs stays balanced. Even though the complete block scheduling algorithm is not yet fully comprehended, as we still need to discern how the scheduler behaves when the number of initial warps (z') in Equation 4 is not constant, we believe that the presented rules lay the foundations for revealing the hidden details of NVIDIA block scheduler.

*2) **Block Scheduling when Shared Memory and Register Usage is Maximum**:* In this section we evaluate the scheduling mechanisms when the dominating term in Equation 1 is the amount of requested shared memory. We performed another experiment in which we first launch a stream S_0 that uses a variable amount of shared memory and then another stream (stream S_1) that does not use shared memory at all. The number of threads/warp per kernel per stream is such to avoid saturation of the capacity of the involved SMs. According to our experiments, the impact on scheduling given by the requested shared memory is considered when a kernel is allocating more than 3KB of shared memory. For shared memory allocations lower than such a threshold, the scheduling mechanism is still dominated by the considerations highlighted in the previous section. Once the kernel in S_0 reaches the 3KB shared memory allocation threshold, our experiments show that the block-to-SM mapping reacts for every 256B increment of the requested shared memory allocation. It is worth mentioning that, 256B matches with the granularity of shared memory allocation that can be found in the file *cuda_occupancy.h*.

Previous work [9], [12] has shown that when shared memory is used in conjunction with multiple streams, such streams are forced to run sequentially. We have found that this issue can be controlled via an advanced CUDA feature, that allows the developer to configure the amount of L1 cache and shared memory that the kernels will use (the *cudaFuncSetCacheConfig* runtime function). If two kernels from different streams are dispatched with different L1/shared configurations, then the kernels belonging to those streams will indeed be forced to run sequentially, even if the actual resource occupancy of the SMs would allow a parallel execution. However, it is sufficient to ensure that the kernels have identical configurations to restore the parallel execution.

Evaluating register pressure as the dominating factor for determining the number of active block in an SM (i.e., the third term of Equation 1) is next to impossible. We observed from experimental results that the impact on scheduling given by the register usage dominates when a kernel is using more than 32 registers (provided that the SM is not already saturated by shared memory allocations or scheduled warps). If a kernel uses less than 32 registers, the scheduling mechanism is still dominated by the effect of shared memory allocations and/or the threads per block configuration. Experimenting with register usage per kernel is difficult: however, it is possible to limit the register usage for kernels by using the *–maxrregcount* compilation flag.

Lesson learned:
The CUDA API lacks features for mapping blocks onto specified SMs. Unfortunately, the CUDA Multi-Process Service (MPS) that allows to partition applications into SMs is not supported on NVIDIA embedded solutions, hence the developer has to avoid the pathological cases described in this section. To this end, we derived a model to understand block-to-SM mapping, as correctly predicting the number of active blocks per SM and how they are partitioned among the SMs is a crucial factor that impacts both performance and predictability when multiple streams are considered. Moreover, the presented findings allow to create smart strategies to schedule multiple CUDA kernels and efficiently utilize the different resources provided by the GPU.

IV. Inside the SM: the warp scheduler

The thread block scheduler pushes thread blocks into the different warp schedulers of each SM. Each SM features two functional units, namely, a certain number of *warp schedulers* and their respective *instruction dispatch units*. The warp scheduler organizes ready-to-execute instructions from a set of available and ready warps, while the instruction dispatch unit forwards instructions to the GPU's SIMD cores. For instance, inside the SM of a Pascal GPU, two warp schedulers and two dispatch instruction units are present. This implies that each warp scheduler is able to launch two different instructions at each clock cycle if these instructions are independent.

Experimental results and publicly available documents [13] show that the warp scheduler used in Maxwell, Pascal, Volta and Turing architectures is the Loose Round Robin (LRR) scheduler [14]. Under this policy, warps are scheduled in a round-robin manner. When a warp reaches an unsatisfied dependency (for instance, a global memory miss), it stalls, so that the next ready warp can be scheduled. This scheduling policy allows hiding memory accesses if there are enough warps ready to execute in order to guarantee fairness among warps.

While the warp scheduling mechanisms remained the same along all NVIDIA GPU generations, an important micro-

Table IV: Load Balancing threshold

#Warps	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
2	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62
3	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61
4	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60
5	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59
⋮																																
31	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32

architectural improvement has been introduced starting from the Volta architecture, i.e., including dedicated FP32 and INT32 ALU pipelines. This allows the instruction dispatching unit to simultaneously execute both floating point and integer operations, so that a kernel can now interleave integer arithmetic with floating-point computations. This aspect becomes crucial when assessing the Worst Case Execution Time (WCET) of a warp, as the presence of operation-type dedicated units dramatically impacts the execution time of the lockstep model. A simple experiment can show the magnitude of such effects. We launch one kernel with a single block composed of 512 threads, so as to saturate the capacity of the ALUs within a single SM. In a first iteration, we force all the 16 warps of the block to only perform operations on double precision variables, and we define the total block execution time as our baseline. Then, 8 out of the 16 warps within the block are changed to perform only integer operations. We perform this experiment on the NVIDIA Jetson TX2 (Pascal architecture) and on the Jetson Xavier (Volta architecture). Results are shown in Figure 6.

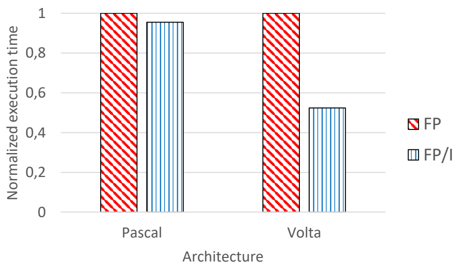


Figure 6: Interleaving of pointer arithmetic with FP computations

The execution time of the single block changes when all the warps perform floating point calculation (FP) and half of the warps perform integer operations (FP/I). The number of warps within the submitted block is such that all the non-specialized ALUs are saturated in the Pascal architecture, hence there is no significant difference in response time when mixing the two kinds of operations. In Volta, floating point ALUs are saturated, but integer specific cores can process the 8 integer-only warps in parallel with the floating point warps, halving the total execution time of the block.

Lesson learned:
One of the issues raised in [12] is that the conclusions drawn by reverse engineering GPU scheduling mechanisms might have to be corrected when future architectures are released. The experiments shown in this section confirms this latest assertion, e.g., the independence of floating point and integer ALUs can indeed play a significant role in WCET estimation of CUDA blocks. The fact that modern NVIDIA GPUs are moving towards architectures characterized by operation-specific logic, as opposed to generic *CUDA cores*, imposes additional considerations when schedulability issues are involved. As an example, consider tensor cores, e.g., a specific logic to compute tensor operations featured in Volta and Turing GPUs. Even if we are unable to show the related experiments for space constraints, we observed that also tensor cores are independent from integer and floating point ALUs.

V. Outside the SM

We would like to explore the factors that influence response time analysis in a CUDA application composed of a plurality of streams. While the previous section focused on the contention within a single SM, this section analyzes the impact of the mechanisms outside the SM where the CUDA blocks under observation are executed.

A. Copy Engine to Execution Engine interference

As highlighted in section III-A, the NVIDIA application scheduler, i.e., the one located in the highest scheduling hierarchy, is designed to have only one application executing on the GPU engines at a given time. This is easily verifiable by dispatching GPU work from different host-side processes [15]. However, the following experiments show that an application can use the copy engine while another application dispatches a kernel in the execution engine. In the experiment depicted in Figure 7, we measured the execution time of a vector add (50MB of data footprint) along with a variable number of processes that perform two different operations using the copy engine: the *memcpy (cudaMemcpy)* and the *memset (cudaMemset)*. Both these memory operations are executed periodically on pinned memory allocations, i.e., by allocating device-only visible memory through *cudaMallocHost* and *cudaMalloc*. Each interfering process executes a *memcpy* or *memset* on a single non-default stream. These experiments have been conducted in the Jetson TX2 and Xavier platforms.

The difference in the response time with respect to the baseline is noticeable. Hence, we can conclude that the NVIDIA application scheduler may co-run different applications, breaking their temporal isolation when requiring different engines. The implications may be substantial, in that the GPUs DMA transfers operated through the copy engine can easily saturate

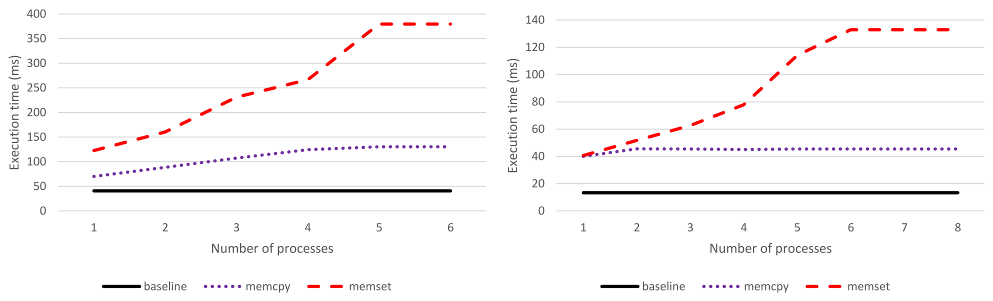


Figure 7: Memory interference caused by EE and CE co-run with a varying number of processes: Jetson TX2 (left) and embedded Xavier (right).

the total available bandwidth of the system RAM. In the Pascal architecture, this causes deterioration in the kernel response time up to almost 3x for *memcpy* operations and up to almost 8x for *memset* commands. Similar conclusions can be drawn for the Volta integrated GPU.

Another experiment considers the memory interference when varying the footprint of the kernel under observation. Specifically, we launch a kernel performing a vector add with a variable memory footprint (from 512KB to 500MB). Then, we add an interfering process performing either a *memset* or a *memcpy* on a fixed size working set (50 MB). The results are shown in Figure 8. As expected, the interference impact is more noticeable at higher footprints.

Lesson learned:
As the NVIDIA scheduler separately arbitrates EE and CE, it is important to take into account the memory interference due to copy operations performed by co-running jobs, be they from the same process, or from different processes.

B. GPCs and shared memory bus

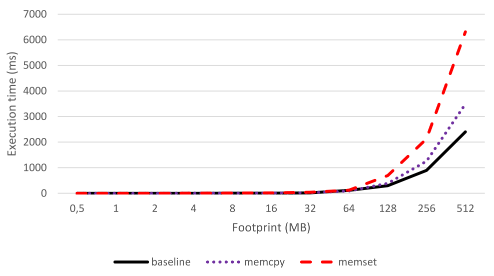
As already highlighted in section II, an NVIDIA GPU is partitioned into SMs, which themselves are grouped into larger clusters called Graphics Processing Clusters (GPC). For instance, the integrated GPU in the Xavier embedded platform features 8 SMs and 4 GPCs. Hence, each GPC is composed of two SMs. The first GPC groups the SMs with ID 0 and 1, the second GPC groups the SMs with ID 2 and 3, and so on. As mentioned in section III-C, the round-robin mechanism forces blocks to be scheduled first in SMs with even IDs and then in those with odd IDs. Previous works [16] focused on the interference produced in the memory hierarchy, however none of these works distinguished between the interference caused by blocks scheduled in the same or in different GPCs. In the following experiment, we performed a vector add on the Xavier platform. The baseline workload executed a single kernel with 4 blocks, utilizing 4 SMs (0, 2, 4 and 6) on 4 different GPCs. This causes each GPC to have one idle SM. The completion time of the kernel block scheduled in SM 0 is shown in the baseline curve of Figure 9, varying the working set size of the vector add. Then, we launched an additional kernel on a different stream, performing another vector add on a single block. This block is scheduled in SM 1, thus causing the first GPC to be fully utilized. The interfering effect on the execution time of the original block in SM0 is shown in the dotted curve of Figure 9. If the interfering kernel is instead launched on a different GPC (e.g., in SM3), the interference is milder, as shown in the dashed curve of the figure.

Note that the interference is not caused by sharing ALUs (i.e., CUDA cores), as blocks are scheduled in different SMs, but it is due to memory contention. The contention is higher in case of intra-GPC interference (up to 10x), since in this case the warps share the same bus for accessing GPU L2 and system RAM.

Lesson learned:
Interference on the GPU memory hierarchy can significantly impact response times of tasks submitting commands to more than one stream. More specifically, the WCET of a single observed CUDA block can increase up to 10x if a memory bound kernel block is scheduled onto the same GPC. If the interfering block runs on a different GPC, contention is still present, although to a lesser extent. Interestingly, by following equations 2-3, the NVIDIA block scheduler initially distributes blocks with the same dimensions in a round-robin fashion, starting with SMs with an even index and then proceeding with the ones with an odd index. In this way, the scheduler mitigates any possible memory interference among SMs that are part of the same GPC.

VI. Related Work

In [17], Jia et al. presented an extensive analysis of the NVIDIA Volta GPU architecture, unveiling many architectural details, such as instruction latencies and warp scheduler details, by means of microbenchmarking. While this work derived important considerations for a schedulability analysis of CUDA-enabled applications, it did not discuss CUDA threads scheduling details and the related predictability pitfalls. In our work, we detailed how scheduling works in NVIDIA GPU-accelerated systems, highlighting the predictability and performance threats given by memory interference (Section V). In [9], an analysis is presented of the scheduling behavior of the NVIDIA Jetson TX2 platform. The authors acknowledge that it is not possible to confirm with certainty the scheduling behavior of the GPU through black-box experimentation. In contrast, we were able to achieve a deeper understanding of



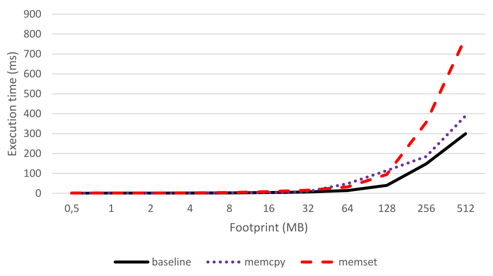


Figure 8: Memory interference memset and memcopy, TX2 (left) and Xavier (right).

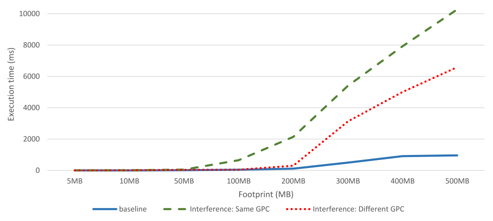


Figure 9: Memory interference at inter/intra GPC level.

the behavior of the GPU scheduler through reverse engineering and by performing simple experiments. This allowed us to correct misstatements or errors found in previously published works. For example, we have shown that the policy adopted by the CUDA thread block scheduler to assign blocks to SMs may be different from the round-robin policy assumed in [4], [18], [19]. Moreover, we shed some light on the hidden and implicit inter-stream synchronization points assumed in [9], [12] when launching multiple CUDA streams with different shared memory requirements. Correctly inferring the details of the block scheduler is important also to correct the many works that propose CUDA stream reordering mechanisms assuming round-robin policies [20], [21], [3].

Existing works motivated GPU concurrency [22], [23] as a means to improve GPU resource utilization and, consequently, throughput. Rodinia [24] and Parboil [25] are well-known benchmark suites for heterogeneous platforms designed to evaluate the performance of different APIs and programming models (e.g., OpenMP, CUDA, OpenCL). In [10], a comprehensive set of measurements is presented to assess the resource utilization of Parboil2 and Rodinia on a Fermi architecture. Architecturally, Fermi (released in 2010) has 32768 registers available per block, 49152 bytes of shared memory per block and a maximum number of 1536 threads per SM. Specifically, Parboil2 benchmarks present an average under-utilization of 40% of threads and blocks, 30% of registers and 80% of shared memory. On the same line, Rodinia under-utilizes 35% the number of threads, 47% on the number of registers, 88% on the amount of shared memory and 52% on the number of blocks that are defined in the kernel invocation. In [26], Karki et al. present a benchmark suite that evaluates different deep neural network (DNN) architectures from a temporal and a spatial perspective. The metrics considered in this analysis are grid dimension, block dimension, shared memory usage and register utilization. Experimental results show that in many cases inference on the networks under-utilizes GPU resources. For instance, SqueezeNet, VGG or CifarNet are described by kernel configurations that are unable to dispatch thread blocks in all the available streaming multiprocessors, hence severely under-utilizing the GPU computing resources. Paras Jain et al. in [27] claim that small inference batch sizes can lead the GPU to low utilization under 15%.

In this regard, most neural network topologies can be modelled as sequences of operations, e.g., sequentially executing inference layers. However, other popular neural network models also feature parallel layers in which precedence constraints are relaxed. As an example, GoogleNet [28] is a convolutional neural network composed of 22 convolutional layers, featuring a large number of inception modules [29]. Each inception module is a convolutional subnetwork that integrates four parallel and independent layers. Unfortunately, most of the APIs for tensor processing, such as Tensorflow or Keras, do not natively support the parallel execution of layers within the same GPU, hence leading to a severe under-utilization. Another example is given by the Apollo Autonomous Driving framework, which can present up to seven instances of DNN-inference work concurrently running [30], e.g., by having a single process dispatching work through many CUDA streams.

VII. Conclusion

In this work, we provided an exhaustive overview of the different arbitration mechanisms that characterize an NVIDIA GPU-accelerated platform. Through extensive experiments we showed that a number of previously accepted assumptions with regards to CUDA streams and block scheduling were incomplete, potentially leading to wrong conclusions. We therefore inferred the different heuristics employed by the hierarchical scheduling scheme adopted by the NVIDIA GPU subsystem, so as to consolidate previously published assumptions. Our experiments showed that a precise understanding on the arbitration mechanisms is of paramount importance for deriving a sound schedulability analysis and to maximize GPU's resource utilization. As future work, we aim at exploiting such results to provide more accurate timing bounds for GPU-accelerated workloads. We also plan to derive stream re-ordering mechanisms that are able to fully utilize GPU parallelism while guaranteeing predictability.

Acknowledgment

This article is part of a joint collaboration between researchers that work on two projects that have received funding from the SPHERE "Software architecture for Predictable HEterogeneous REal-time systems" project (PRIN 2017, CUP: E94I19000820001) and the Chair for Cyber-Physical Systems in Production Engineering at TUM.

References

[1] T. M. John Cheng, Max Grossman, *Professional CUDA C Programming*, 1st ed. Birmingham, UK, UK: Wrox Press Ltd., 2014.

[2] I. S. Olmedo, N. Capodieci, and R. Cavicchioli, "A perspective on safety and real-time issues for gpu accelerated adas," in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, Oct 2018, pp. 4071–4077.

[3] M. Awatramani, J. Zambreno, and D. Rover, "Increasing gpu throughput using kernel interleaved thread block scheduling," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct 2013, pp. 503–506.

[4] H. Li, D. Yu, A. Kumar, and Y. Tu, "Performance modeling in cuda streams — a means for high-throughput data processing," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 301–310.

[5] T. S. Greg Ruetsch, Paulius Micikevicius, "Optimizing Matrix Transpose in CUDA," NVIDIA, Tech. Rep., 2010.

[6] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for gpu with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 119–130.

[7] N. Capodieci, R. Cavicchioli, and M. Bertogna, "Work-in-progress: Nvidia gpu scheduling details in virtualized environments," in *2018 International Conference on Embedded Software (EMSOFT)*. IEEE, 2018, pp. 1–3.

[8] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "Sigamma: server based integrated gpu arbitration mechanism for memory accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. ACM, 2017, pp. 48–57.

[9] 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)*, Dec 2017, pp. 104–115.

[10] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," *SIGPLAN Not.*, vol. 48, no. 4, pp. 407–418, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451160

[11] V. Volkov, "Better performance at lower occupancy," *Proceedings of the GPU Technology Conference, GTC*, vol. 10, 2015.

[12] 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)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 20:1–20:21. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8984

[13] P. Voudouris, "Analysis and modeling of the timing behavior of gpu architectures," 2014.

[14] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 163–174.

[15] V. Madumbu, "Adas/ad challenges:gpu scheduling & synchronization," GTC, San Jose, 2017.

[16] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms," in *22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*, 2017.

[17] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[18] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, June 2014.

[19] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Preemptive thread block scheduling with online structural runtime prediction for concurrent gpgpu kernels," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014, pp. 483–484.

[20] R. A. Cruz, C. Bentes, B. Breder, E. Vasconcellos, E. Clua, P. M. de Carvalho, and L. M. Drummond, "Maximizing the gpu resource usage by reordering concurrent kernels submission," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 18, p. e4409, 2019, e4409 cpe.4409. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4409

[21] F. Wende, F. Cordes, and T. Steinke, "On improving the performance of multi-threaded cuda applications with concurrent kernel execution by kernel reordering," in *2012 Symposium on Application Accelerators in High Performance Computing*, July 2012, pp. 74–83.

[22] S. . Shekofteh, H. Noori, M. Naghibzadeh, H. Fröning, and H. S. Yazdi, "ccuda: Effective co-scheduling of concurrent kernels on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 766–778, April 2020.

[23] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou, "Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 208–220.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[25] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," 2012.

[26] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite for various accelerators," *CoRR*, vol. abs/1901.04987, 2019. [Online]. Available: http://arxiv.org/abs/1901.04987

[27] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica, "Dynamic space-time scheduling for GPU inference," *CoRR*, vol. abs/1901.00041, 2019. [Online]. Available: http://arxiv.org/abs/1901.00041

[28] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[29] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 2818–2826.

[30] R. Pujol, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.