# Interference analysis of shared last-level cache on embedded GP-GPUs with multiple CUDA streams

Gianluca Brilli, Paolo Burgio
*University of Modena and Reggio Emilia, Italy*
{gianluca.brilli, paolo.burgio}@unimore.it

***Abstract*—In modern heterogeneous architectures, the access to data that the application needs is a key factor, in order to make the compute task efficient, in terms of power dissipation and execution time. The new generation SoCs are equipped with large LLCs, in order to make data access as efficient as possible. However, these systems introduce a new level of complexity in terms of the system's predictability, because concurrent tasks must compete for the same resource and contribute to generate interference between them. This paper aims to provide a preliminary qualitative analysis in terms of interference degree that is generated when several concurrent streams are in execution, for example one that performs useful computing tasks and one that generates interference. Specifically, we tested two important primitives: *vadd* and *gemm*, respectively subjected to interference with: *i)* a concurrent kernel that performs read from shared memory. *ii)* concurrent stream that performs host-to-device memory copy.**

***Index Terms*—GP-GPUs, embedded systems, Real-Time systems, caches**

## I. Introduction, and motivation for this work

The increasing demand for high-performance computational capabilities at low size-weight and power (SWaP) of modern embedded systems paved the way to the adoption of heterogeneous computing platforms with multi-core host and many-core accelerators. Especially, integrated GPGPUs (iGPUs) [6], [7] are today's preferred to other acceleration paradigms, e.g., based on FPGAs or application-specific integrated circuits (ASICs), in applications with data-parallel workloads, such as computer vision and AI systems employing deep neural networks. This is the case of advanced automotive systems[1], where AI/DNN are increasingly being adopted as reference for building partly- or fully- automated vehicles of tomorrow. Unfortunately, these systems demand not only for high peek performance, but also –and especially– worst case performance, and the increased architectural complexity of modern iGPUs makes it extremely cumbersome to perform an effective non-pessimistic worst-case timing analysis of system. Recently, researchers [2], [4], [8] proved that the main source of unpredictability in such systems are contentions on shared resources, such as memory banks, but yet only few works [3] focused on shared GPU last-level cache (LLC) which also is a major source of contention, *that affects both host and accelerator complexes*. The reason for this lack of material is that, hardware providers (in this case, NVIDIA) are too often reluctant to disclose the internals of their highly-optimized architectures and memory drivers, forcing researcher to a huge effort of reverse engineering for understanding them [3]. This is also interesting because *last-level-caches are the closest shared resources between cores*, hence they are affected by the whole memory traffic due to local caches misses, and deserve a special attention.

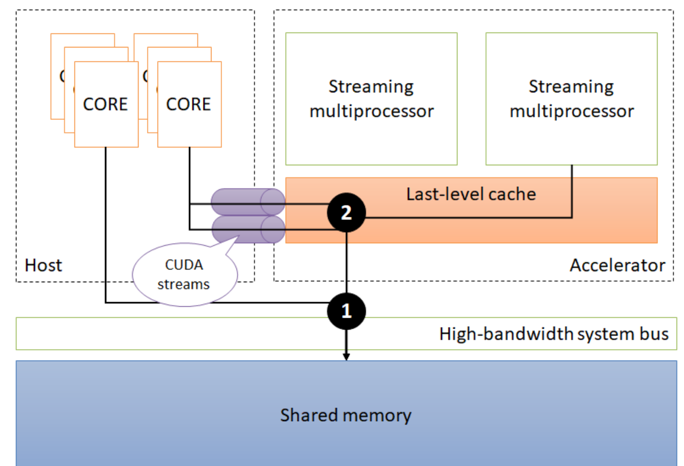


Fig. 1. Reference iGPU architecture with key architectural bottlenecks

**CUDA streams.** One of the main performance booster, when adopting a host-accelerator paradigm, is the possibility of overlapping multiple computation kernels and data transfers between the CPU and the GPU. In NVIDIA GPGPUs, this is possible thanks to the abstraction of *CUDA streams*, where both execution and data transfer request are issued from the application control running on the host. Unfortunately for RT engineers, CUDA streams introduce an additional level of parallelism, further increasing system complexity, and we will show how the complex mechanism for stream management implemented in platform drivers *enables an additional source of contention in the system*, negatively affecting predictability, because they create interference not only on the shared memory, but also on last-level cache. Circles with numbers in Figure 1 highlight the two main contention points (LLC and memory) in the considered system.

**Platform modeling in industry.** Another issue stems from the fact that industrial-grade frameworks for software development, such as Amalthea [1] for the automotive domain, too of-

[1]Figure 1 shows a simplified block diagram of a NVIDIA TX2, where an esa-core host shares memory banks with two CUDA streaming multiprocessors (SM) of the Pascal family.

ten rely on simplified platform model, practically inapplicable and ineffective with the complex structure of iGPUs. Indeed, there is no standard approach to modeling both the implicit memory contention between host cores and GPU cores. Of course, the situation gets even worse when CUDA streams are included in the picture. Indeed, this year's WATERS challenge only focuses on single-stream applications.

Our work wants to be the first one in analyzing and modeling, not only analytically but also with empirical evidence, the contention on LLC introduced by the adoption of multiple CUDA streams.

The rest of the paper is structured as follows: in section II, we provide some ideas regarding the implementation of the interference benchmarks and some expected results. Section III shows the graphs obtained by running the experiments described in Section II on top a representative embedded iGPU, the NVIDIA Tegra X2 and details the results. Finally in Section IV we summarize the experiments carried out in this research and we conclude the paper.

## II. Implementation Details

In this section we provide some details related to the benchmark suite used in the experiments. In particular we decided to measure the slowdown due to the interference on LLC on two basic operations of Linear Algebra, respectively the sum of vectors and the multiplication of matrices ( *vadd* and *gemm* ). From the point of view of the experiments carried out, we quantified the level of cache interference by measuring the performance of the aforementioned kernels with and without interference, in terms of execution time, because unfortunately the metric *l2_l1_read_hit_rate* is not implemented on our target platform. In this way we compared the execution times of the two compute-kernels with and without interference. The slowdown obtained, is obviously due to the LLC data replacement and the consequent high cost of access to the *Shared Memory* outside the chip.

In terms of benchmarks, we decided to test the following two cases:

*i) Compute Kernel and Interference Kernel*: in this first test case we performed a comparison between a compute kernel running on a single *Streaming Multiprocessor (SM)* in isolation (ie. without interference), compared to the same kernel in a concurrent execution with a kernel that reads data from memory, running on a parallel stream, mapped to the second SM inside the Tegra X2 SoC.

```
while (runs --) {
  idx = threadIdx.x * stride;
  while (idx < n) {
    w[idx] = r[idx];
    idx    += blockDim.x;
  }
}
```

Listing 1. Interference kernel implementation.

In listing 1 we can see the simple idea behind the interference kernel used in this work. In particular we can notice the *stride* parameter, which determines the access pattern of the threads to the LLC.

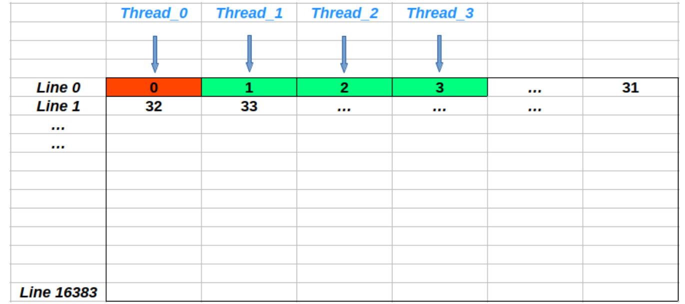


Fig. 2. Thread access pattern to LLC. The case reported in this figure is a *coalesced* access, that we have when the *stride* is equal to 1. In this case only the first thread causes a cache miss (the red cell).

In figure 2 we can see what happens inside the GPU LLC from a graphical point of view, varying the stride. In particular if it is less than the cache line size ($< 32$) we expect the generated interference is ”relatively low”, because some threads will perform cache hit on the LLC. Similarly, by increasing this parameter beyond the LLC line size ($> 32$), we experimented a similar behavior, in which some cache lines will not be affected by the interference memory accesses. Finally in the case of strided accesses of the same size as the cache line ($= 32$), we have the maximum possible interference that we can generate involving a single SM [2], then in this last case, each thread could potentially perform a cache miss, triggering a load from the main memory.

*ii) Compute Kernel and Copy Engine*: in this last comparison we focused on the behavior generated by concurrently launch a compute-kernel, running on top of one of the two SMs, together with a concurrent CUDA-stream that takes care of performing memory copy *host-to-device*, through the *Copy Engine*. This type of copy, moves the target memory, which is stored in an area addressable by the Host complex, inside a region that is visible by the GPU, and also prefetches the data needed on the GPU LLC, in order to bring this data closer to the *Compute Engines*. Within our benchmarks, these memory copies involve all or some parts of cache lines, in particular the cache regions are kept under interference throughout the computation time of the concurrent kernel.

```
while(runs --) {
  cudaMemcpyAsync(w_gpu, r_gpu, LINE_SIZE *
     cache_lines, cudaMemcpyHostToDevice, s1);
  cudaStreamSynchronize(s1);
}
```

Listing 2. Concurrent memory copy stream.

Listing 2 shows in detail the benchmark idea described above. In particular we used the *CudaMemcpyAsync*, for executing memory copy on top a concurrent stream, different from the default one. Through the *cache_line* parameter we can control the number of cache lines under interference, ranging from one single line up to the whole cache size.

[2]The maximum number of threads that can be launched in a block is hardware-limited to 1024 threads, summing each dimension of the grid.

## III. Experimental Results

In this section we report the experimental results from two kernels in isolation, compared to a concurrent execution with the two aforementioned interference cases. The target computing platform is the NVIDIA Tegra X2 [6], a *System-on-Chip (SoC)* that embeds an esa-core processor, in a Big.SUPER configuration, composed of four ARMv8 Cortex A57 and two proprietary NVIDIA Denver. The GPU is composed of two Streaming Multiprocessors (SMs) belonging to the *Pascal* family, summing up $256$ CUDA Cores @ $854-1465$MHz in total. As mentioned before, we used one SM to perform useful computation and the other one for doing memory interference. Regarding the memory hierarchy, we have $8$ GB of LPDDR4, shared between the CPU cluster and the GPU, with a bus width of $128$ bits and a bandwidth of $58.4$ GB/s. Regarding the last level cache system we have a L2 cache, shared between the host complex and a L2 cache for the *Pascal* GPU, the second one is composed of $16384$ cache lines of size $32$ bytes each, summing up $512$KB in total.

In figure 3 we can see a comparison of the execution time needed by the *vadd* kernel, respectively with and without interference. In particular from this first benchmark we experimented an execution time of about $5$ seconds for the baseline version (without interference). Subsequently by launching the interference kernel and varying the *stride* parameter we can notice a linear growth in terms of execution time, until reaching the dimension of the LLC line-size. In this case we have reached the maximum number of possible cache misses and the consequent loads from the main memory, which we can generate with a single SM, composed of one CUDA block of $1024$ concurrent threads. The increase in execution time experimented is proportional to a $6\times$ factor from the baseline version. Further increasing the *stride* parameter even beyond the LLC line-size we have a reduction of the interference factor, up to about $2\times$ compared to the baseline version. Specifically, we observed that the execution time seems to converge towards a specific value. The decrease in execution time is due to the fact that increasing the *stride* parameter beyond the line size of the LLC, we are indeed skipping some cache lines and consequently generating less interference.

In figure 4 we can see a comparison of the same compute kernel, with a concurrent execution of a CUDA stream that performs memory copy *host-to-device*, exploiting the *Copy Engine*. The comparison is always measured in terms of execution time and in this case it is carried out by gradually increasing the number of cache lines affected by the memory copy stream, ranging from a single cache line, up to $16384$ cache lines (the full size of the LLC). In this case we have an increase of about a $1.2\times$ factor in the case of a single cache line interfered, up to a factor of $2.4\times$ when we have interference on the whole cache.

The same test modalities are shown in figure 5 and 6, in particular in figure 5 we can see the growth in terms of execution time due to the interference generated by the concurrent kernel in execution on the second SM. Also in this case we can notice a linear growth of the execution time, by increasing the *stride* parameter, until we reach the line-size of the LLC ( 32-bytes ). In case of the *gemm* task we have an increase of about a $3\times$ factor of the execution time respect to the baseline reference. Beyond this point, also in this case, follows a decrease in the total execution time, until reaching a fixed value, similar to the baseline case.

Finally in the last proposed comparison, shown in figure 6, we note that the effect of interference due to memory copy *host-to-device* does not have a particularly interference impact on the total execution time, unlike what happens in the case of concurrent kernels.

## IV. Conclusion

In this paper we presented, through a sequence of benchmarks, some interference effects on *Last-Level Cache (LLC)*, due to the execution of an interference kernel, mapped on a concurrent *Streaming Multiprocessor (SM)* and by means of memory copy *host-to-device*. The tests were carried out on a device that nowadays can be considered as the state of the art regarding high performance iGPU, the NVIDIA Tegra X2 SoC. Through the analysis of the results obtained we were able to infer the size of the single cache line, a fundamental parameter to be known if we intend to develop embedded software that makes efficient use of caches. We also highlighted that interference on LLC, in case of kernels running on separate SMs, has a very high impact on the number of accesses to the main memory, which translates into *i)* a considerable increase in total execution time, equal to a factor of about $6\times$ in the case of the *vadd* and about $3\times$ for the *gemm*. *ii)* difficulty in making predictable the *worst-case execution time (WCET)*. This justifies and supports the need for adopting predictable models, in which several tasks must synchronize the memory access phases.

## References


[1] Amalthea Consortium. Amalthea. model based open source development environment for automotive multi core systems, 2014.

[2] 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, Limassol, Cyprus, September 12-15, 2017*, pages 1–10, 2017.

[3] B. Forsberg, L. Benini, and A. Marongiu. Taming data caches for predictable execution on gpu-based socs. In *DATE19 to appear*, 2019.

[4] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET derivation under single core equivalence with explicit memory budget assignment. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, pages 3:1–3:23, 2017.

[5] X. Mei and X. Chu. Dissecting gpu memory hierarchy through microbenchmarking. 2017.

[6] NVIDIA. Jetson TX2 Module, 2017.

[7] NVIDIA. Jetson AGX Xavier Developer Kit, 2018.

[8] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Trans. Computers*, 65(2):562–576, 2016.

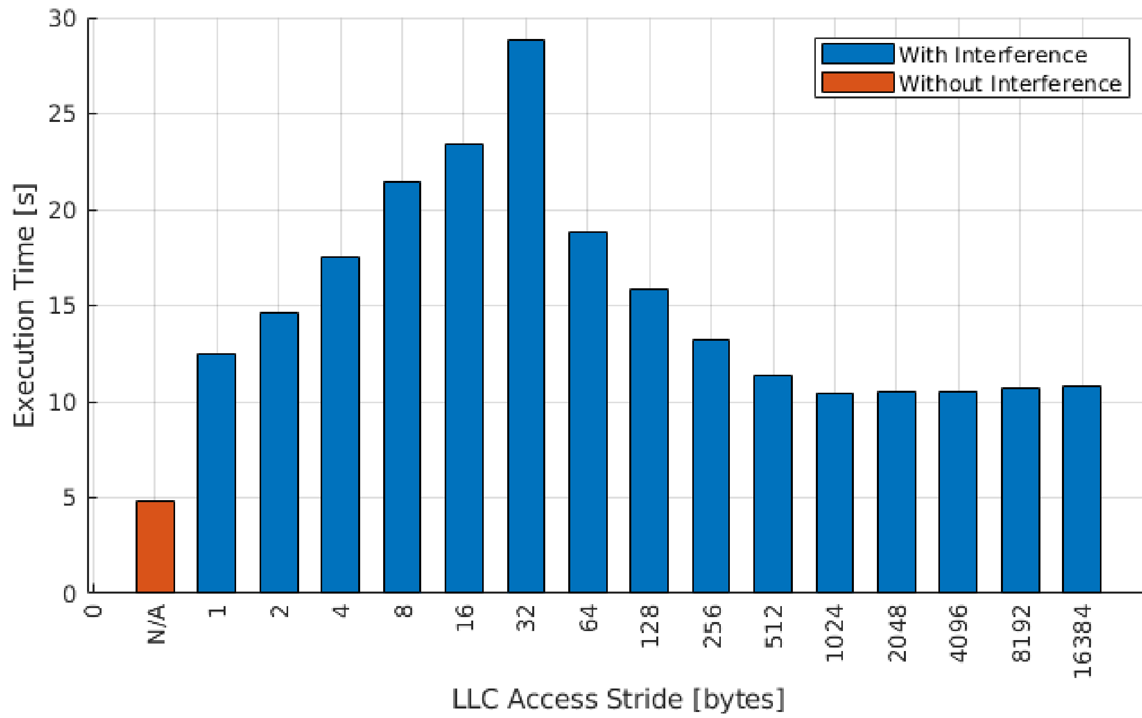


Fig. 3. Comparison in terms of execution time of the *vadd* kernel in isolation (red bar), compared to the same kernel running concurrently with an interference kernel (blue bars).

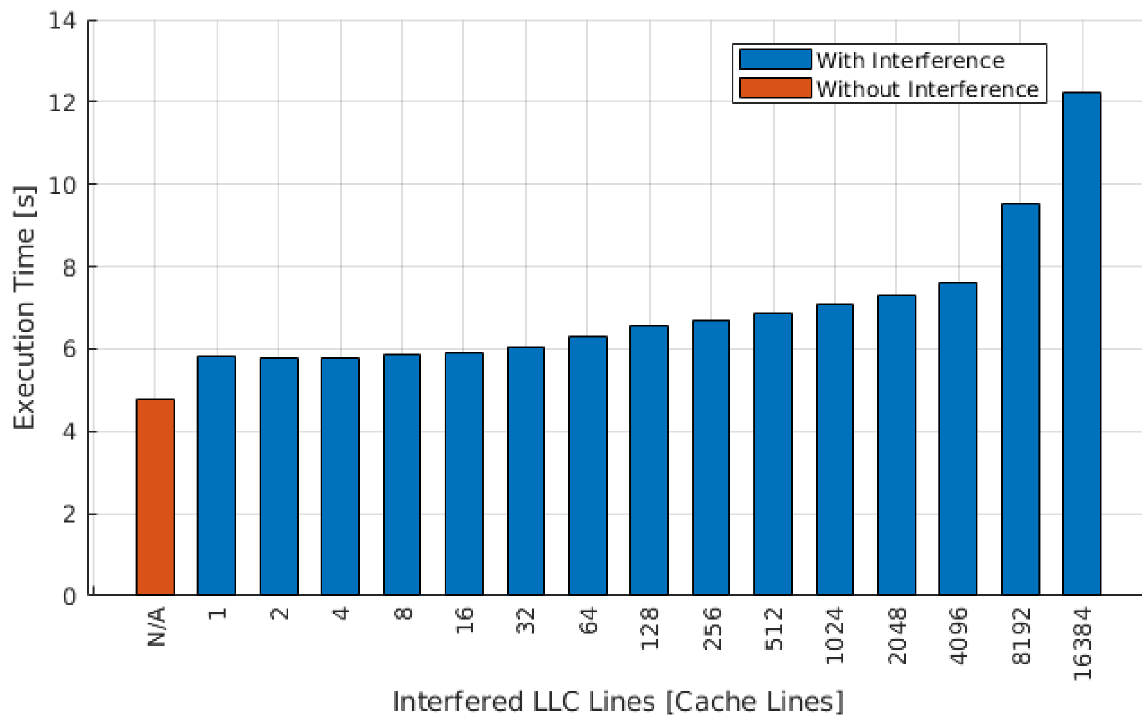


Fig. 4. Comparison in terms of execution time, between the *vadd* kernel in isolation (the red bar) and the same kernel in execution with a concurrent stream that performs memory copy *host-to-device* (the blue bars).

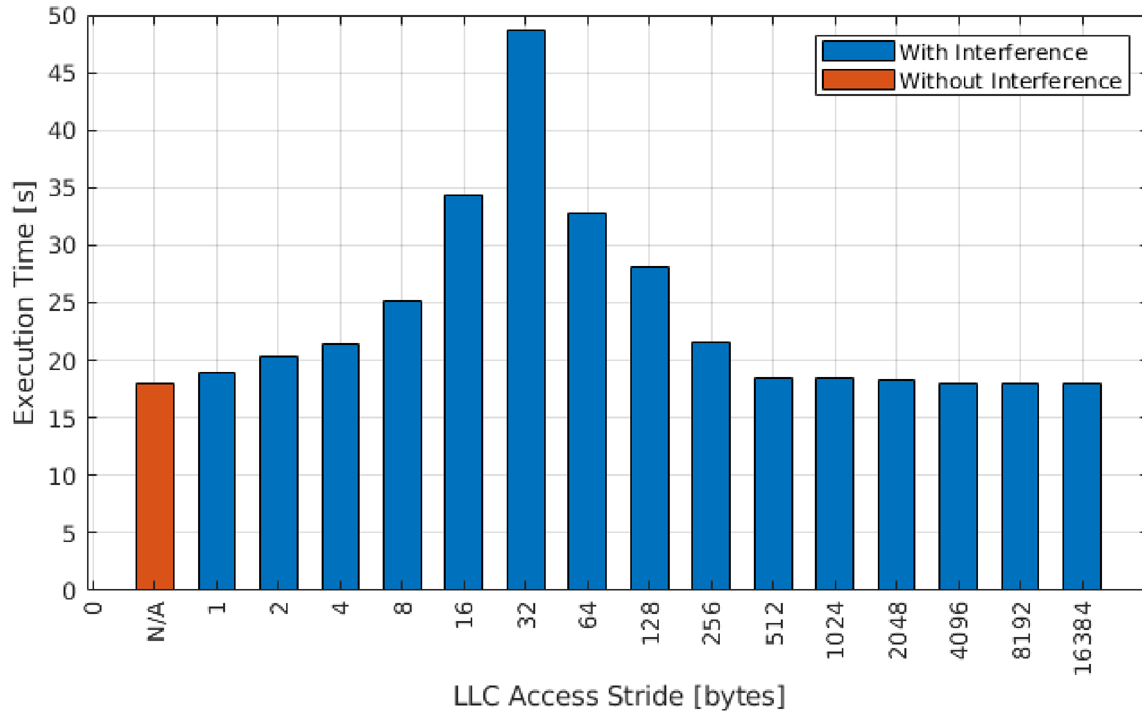


Fig. 5. Comparison in terms of execution time of the *gemm* kernel in isolation (red bar), compared to the same kernel running concurrently with an interference kernel (blue bars).

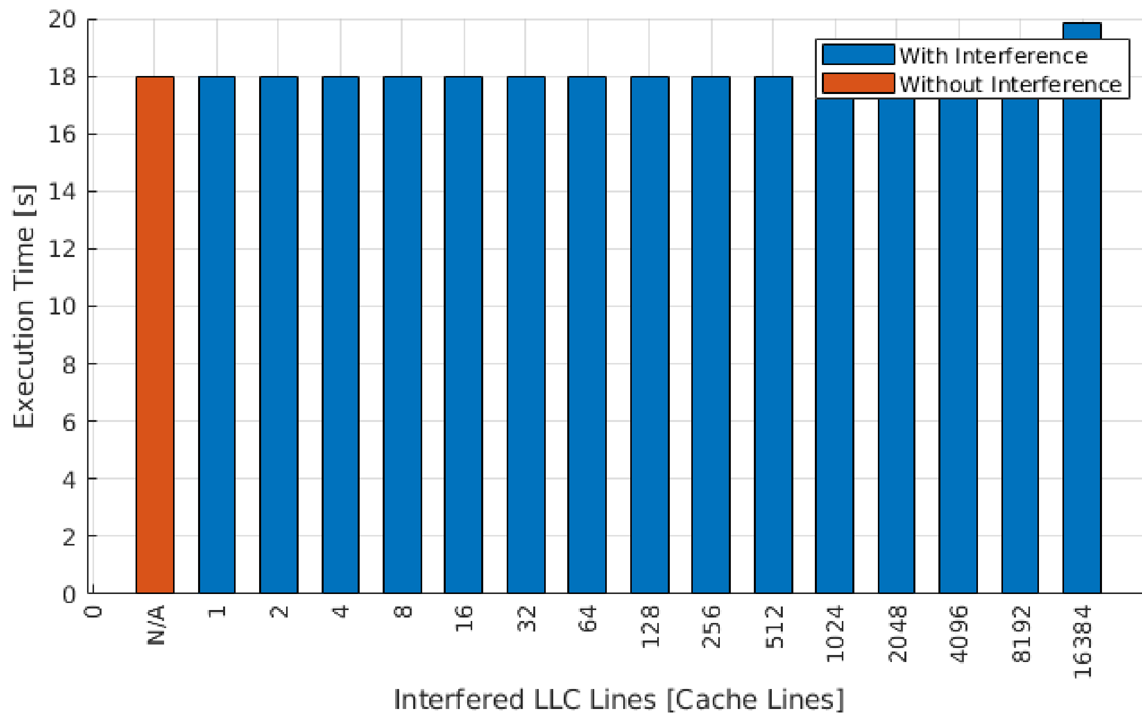


Fig. 6. Comparison in terms of execution time, between the *gemm* kernel in isolation (the red bar) and the same kernel in execution with a concurrent stream that performs memory copy *host-to-device* (the blue bars).