Predicting GPU kernel's performance on upcoming architectures

Lucas Van Lanker, Hugo Taboada, Elisabeth Brunet, François Trahay

Predicting GPU kernel's performance on upcoming architectures

Lucas Van Lanker[1,3], Hugo Taboada[1,2], Elisabeth Brunet[3] and François Trahay[3]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
{lucas.vanlanker,hugo.taboada}@cea.fr
[2] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France
[3] Télécom SudParis, Institut Polytechnique de Paris, Inria, 91000 Évry, France
{elisabeth.brunet,francois.trahay}@telecom-sudparis.eu

Abstract. With the advent of heterogeneous systems that combine CPUs and GPUs, designing a supercomputer becomes more and more complex. The hardware characteristics of GPUs significantly impact the performance. Choosing the GPU that will maximize performance for a limited budget is tedious because it requires predicting the performance on a non-existing hardware platform.
In this paper, we propose a new methodology for predicting the performance of kernels running on GPUs. This method analyzes the behavior of an application running on an existing platform, and projects its performance on another GPU based on the target hardware characteristics. The performance projection relies on a hierarchical roofline model as well as on a comparison of the kernel's assembly instructions of both GPUs to estimate the operational intensity of the target GPU.
We demonstrate the validity of our methodology on modern NVIDIA GPUs on several mini-applications. The experiments show that the performance is predicted with a mean absolute percentage error of 20.3 % for LULESH, 10.2 % for MiniMDock, and 5.9 % for Quicksilver.

Keywords: Performance projection · GPU architecture · Roofline model.

1 Introduction

Designing a supercomputer is a complex task that requires balancing multiple properties including the price of components, and their performance. GPUs are a major part of the design space to explore, as new generations of GPUs deliver ever-better performance. Due to the high price of high-end GPUs, estimating the performance of an application on a target GPU architecture is crucial before committing to buy new hardware.

Predicting GPU applications' performance is quite challenging: GPUs rely on many cores and a complex memory hierarchy, vendors may use closed source documentation that hides artifacts such as memory bank conflict or code divergence, and applications require fine tunings to use the GPU to its full potential.

As we will see in the related work section, to project performance, several approaches exist, such as simulation, statistical model with machine learning, and analytical model. However, they are either too much time-consuming, or they cannot be applied to large applications. In this context, less accurate but faster methods such as performance projection models become interesting.

In this paper, we propose a methodology for predicting the performance of kernels running on a target GPU, as presented in Figure 1. The proposed method runs a program on an existing GPU once, and it projects the application performance on a target GPU. With a single sample run, we characterize the application behavior according to different criteria such as its computational intensity, data placement, and other properties that can be measured, analyzed, and modeled. Our methodology compares the assembly code of the application on both the source GPU and the target GPU and uses the model in order to project the application performance on the target GPU.

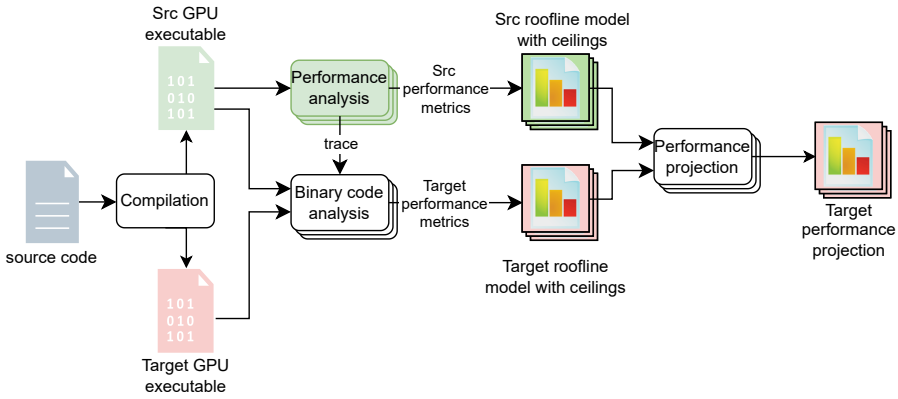


Fig. 1: Summary of the proposed projection workflow.

The contributions of this paper are the following:

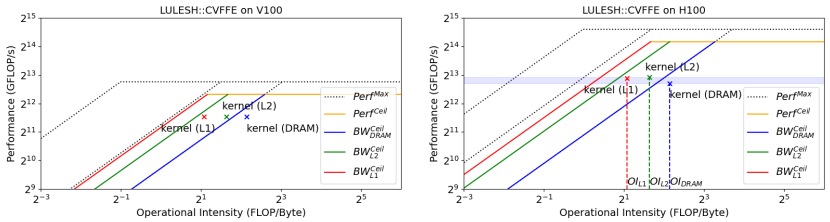
- We propose a kernel performance projection methodology for GPUs from the run of a kernel on a given source GPU to a target one based on roofline models of GPUs with kernel-specific ceilings;
- We implement the methodology for NVIDIA GPUs. Our implementation combines a comparison of the assembly-code of both GPUs with a performance projection on both roofline models;
- We validate our methodology by projecting multiple mini-applications (Hydro1d, UVMBench, Quicksilver, LULESH, miniMDock) running on V100 GPUs to several modern NVIDIA GPUs like A100 and H100. In this evaluation, our methodology achieves a mean absolute percentage of error (MAPE) comprise between 10.3 % and 17.0 %.

The remainder of this paper is organized as follows. In Section 2, we describe how to modify a roofline model to take into account the characteristics of a

kernel. In Section 3 we describe how we predict the performance of a kernel that would run on a target GPU. We present the implementation details of our methodology in Section 4. Section 5 presents the experimental evaluation of our implementation. We discuss related work in Section 6. Finally, we conclude the paper in Section 7.

2 Roofline model of a GPU with kernel-specific ceilings

As introduced in [21], the roofline model of a machine gives its upper bound in terms of performance and memory bandwidth. These bounds can be estimated based on the hardware characteristics($Perf^{Peak}$ and BW^{Peak}), or by measurement with benchmark applications such as HPL [17] and STREAM [14] ($Perf^{Max}$ and BW^{Max}). Depending on an application operational intensity (OI) and its performance, the roofline model indicates the optimization level of the application and its limiting hardware component. Several works have extended the roofline model for GPUs [6, 12, 22–24]. In particular, the NVIDIA Nsight Compute (*ncu*) profiler [16] defines a hierarchical roofline which relies on the OI of all GPU's cache levels (eg. BW_{L1}^{Max}, BW_{L2}^{Max}, BW_{DRAM}^{Max}).



(a) On source GPU V100.

(b) On the target GPU H100 with its performance projection (blue area).

Fig. 2: Roofline models with ceilings for the LULESH CVFFE kernel.

The roofline model describes the maximum attainable performance on a given machine for a given operational intensity, as depicted by Equation 1.

$$roofline(OI) = min(BW^{max} \times OI, Perf^{max}) \tag{1}$$

However, this upper bound to performance could only be reached with a perfect compute efficiency and memory efficiency, which is unrealistic. To better understand how a given kernel performs, we refine the roofline model by adding ceilings of the compute and memory capacities. These ceilings are noted $Perf^{Ceil}$, BW_{L1}^{Ceil}, BW_{L2}^{Ceil}, and BW_{DRAM}^{Ceil} in Figure 2.

Compute efficiency. Roofline models usually model GPU maximum performance $Perf^{max}$ by relying only on Fused Multiply-Add (FMA) instructions performance. Nevertheless, the compute efficiency of a kernel is dependent on its

floating-point operation mix [8, 22]. Indeed, kernels use other floating-point instructions such as ADD, or MUL. In Equation 2, we enhance the roofline $Perf^{max}$ term of Equation 1 by averaging the maximum performance of both FMA and ADD+MUL operations as summed in Equation 2 :

$$\begin{aligned} Perf^{mix}(k) = Perf^{max}_{FMA} \times \frac{N_{FMA}}{N_{FMA} + N_{ADD} + N_{MUL}} \\ + Perf^{max}_{ADD\_MUL} \times \frac{N_{ADD} + N_{MUL}}{N_{FMA} + N_{ADD} + N_{MUL}} \end{aligned} \tag{2}$$

where N_{FMA} (respectively ADD, MUL) corresponds to the number of FMA instructions in the studied kernel k, and $Perf_{FMA}$ (resp. ADD_MUL) is the maximum performance of only FMA instructions.

To go further, we also take into account the GPU warp usage, which is the mean number of active threads per warp instruction divided by the size of a warp, which gives the performance ceiling of $Perf^{ceil}(k)$ the kernel k as described in Equation 3, and represented with a yellow horizontal line in Figure 2a.

$$Perf^{ceil}(k) = \frac{active\_thr\_per\_instr}{warp\_size} \times Perf^{mix}(k) \tag{3}$$

Memory performance. A kernel performance may also be limited by its memory performance. A kernel accesses data located in various place of the memory

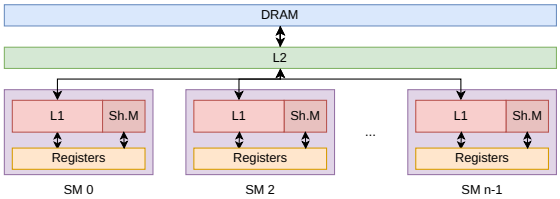


Fig. 3: Memory hierarchy of a GPU.

hierarchy, as illustrated in Figure 3, where the DRAM is the main memory of the GPU, L2 a coherent cache level shared by all streaming multiprocessors (SM), L1 a cache private to each SM, and the shared memory (Sh.M), which is a fast memory shared by all the threads of a CUDA block.

We assume that the time for getting data from a memory location depends on the memory location bandwidth, as described in Equation 4.

$$t_{mem}(k) = \frac{N_{mem}(k)}{BW^{max}_{mem}} \tag{4}$$

where N_{mem} is the number of bytes accessed by the kernel that hits the memory level mem, and BW^{max}_{mem} its maximum bandwidth as used in the roofline model.

Furthermore, we model the shared memory differently, as it is physically located in the L1 cache in our studied GPUs which is divided in memory banks.

The time for getting data from the shared memory (t_{ShM}) depends on the bank conflicts and the number of clock cycles needed to handle the memory requests. To be optimal, the number of accessed bytes per clock cycle (N_{ShM_pc}) should be 128 ($N^{max}_{ShM\_pc}$), due to the 32 banks, each with a bandwidth of 4 bytes per clock cycle [15], hence our definition of t_{ShM} in equation (5).

$$t_{ShM}(k) = \frac{N_{ShM}(k)}{N_{ShM\_pc}(k)} \times \frac{N^{max}_{ShM\_pc}}{BW^{max}_{ShM}} \tag{5}$$

Then we update the roofline model with bandwidth ceilings for L1, L2, and DRAM (BW^{ceil}_{L1}, BW^{ceil}_{L2} and BW^{ceil}_{DRAM} in Figure 2a). For each memory location, the bandwidth ceiling takes into account all the data access that traverse the memory location, e.g., L2 bandwidth ceiling is the bandwidth for the data coming from both the L2 cache, and the DRAM. Overall these bandwidth ceilings are computed as a weighted harmonic mean of bandwidths, as described by Equations 6, 7, and 8.

$$BW^{Ceil}_{L1}(k) = \frac{N_{L1}(k) + N_{ShM}(k) + N_{L2}(k) + N_{DRAM}(k)}{t_{L1}(k) + t_{ShM}(k) + t_{L2}(k) + t_{DRAM}(k)} \tag{6}$$

$$BW^{Ceil}_{L2}(k) = \frac{N_{L2}(k) + N_{DRAM}(k)}{t_{L2}(k) + t_{DRAM}(k)} \tag{7}$$

$$BW^{Ceil}_{DRAM}(k) = \frac{N_{DRAM}(k)}{t_{DRAM}(k)} \tag{8}$$

where $N_{mem}(k)$ is the number of memory access to the memory location mem performed by kernel k.

3 Projecting the roofline model with ceilings to a target GPU

In order to predict the performance of a kernel on a target GPU, we first build its roofline model with ceilings on both source and target GPU following the method described in Section 4.2, as illustrated in Figure 2.

Then, we measure the performance of the kernel itself $Perf^{Meas}_s(k)$ on the source GPU, as well as its three OIs ($OI_{L1}(k)$, $OI_{L2}(k)$, and $OI_{DRAM}(k)$), before projecting it on the target GPU using formula (9) which is presented by C. Gavoille et al. in [8] and Kwack et al. in [13].

The idea is to consider for each OI the ratio between the measured performance of the kernel $Perf^{Meas}_s(k)$ and the corresponding roofline $roofline_{s_{mem}}(OI_{s_{mem}}, k)$ on the source GPU, and to project this ratio by using the corresponding roofline and the OI of the target GPU $roofline_{t_{mem}}(OI_{t_{mem}}, k)$.

$$Perf_t(mem, k) = \frac{Perf_s(k)}{roofline_{s_{mem}}(OI_{s_{mem}}, k)} \times roofline_{t_{mem}}(OI_{t_{mem}}, k) \tag{9}$$

Thus, we obtain three performance values $Perf_t(L1, k)$, $Perf_t(L2, k)$, and $Perf_t(DRAM, k)$ which give an interval of performance to expect, as illustrated by the blue area in Figure 2b. Finally, we compute the kernel execution time by dividing $FLOP$ by the performance interval.

4 Implementation

We implemented our performance projection methodology for NVidia GPUs. This Section details the metrics that are collected when characterizing a kernel and a GPU. We also describe how we analyze the assembly instructions of a kernel in order to estimate its OIs.

4.1 Collecting metrics

Our performance projection methodology requires to gather capabilities of both target and source GPUs. Reminding that the target GPU is unavailable, the theoretical peak $Perf_t^{Peak}$ and BW_t^{Peak} performance can be retrieved from the target GPU specification. The measured $Perf_t^{Max}$ and BW_t^{Max} performance are either obtained thanks to publicly available benchmarks results, or estimated if the $Peak/Max$ ratio is assumed to be similar for the source GPU and the target GPU. Source GPU capabilities are actually measured. We collect the maximum performance $Perf_s^{Max}$ using the High Performance Linpack (HPL) [17] benchmark, and the peak bandwidth BW_s^{Max} using the STREAM bandwidth [14] for each memory level. Application-specific performance data are also retrieved for each kernel during a profiling execution. We run the application with the *ncu* profiler and collect several metrics for each kernel, such as the execution time, the number of bytes accessed at each memory level, and the number of FLOPs.

4.2 Estimating the target operational intensity

As depicted in Figure 2a, we first characterize the actual kernel performance on the source GPU obtained by a profiling run. During this run, the amount of data accessed through the L1, L2, and DRAM are also gathered and we compute three operational intensities of the kernel on the source GPU. Following Equation (10), the OI for a memory level is defined as the number of FLOPs per byte written or read at this cache level, with N_{mem}^{+} the number of bytes traversing the memory level mem.

$$OI_{mem}(k) = \frac{FLOP(k)}{N_{mem}^{+}(k)} \tag{10}$$

Now, in order to accurately project the kernel performance on a target GPU, it is necessary to estimate the kernel memory usage, and its OIs on the target GPU. While the memory usage is considered as roughly similar from one GPU to another in our current implementation, our model takes in consideration the fact that an OI may differ because both GPUs may provide different instruction sets. We estimate the target OI by analyzing the kernel assembly instructions (SASS)

of the target machine binary, and comparing it with SASS instructions traces we profiled on the source GPU. In fact, since the intermediate PTX language of a kernel is the same for all GPUs, we only need to compare instructions that implement special functions, e.g. divisions or logarithms, which will be translated into multiples different floating operations depending on the GPU [1,24] and that the other instructions are executed the same way for both machines. Thus, we compare the basic blocks of the control-flow graph for both GPUs that contain such special functions: for these blocks, the SASS traces of the source GPU give the number of active threads, and we assume that the same number of threads are active in the equivalent blocks in the SASS instructions of the target GPU.

5 Experiments

In this Section, we evaluate our performance projection methodology. For this purpose, we run 5 mini-applications, i.e. Hydro1D, UVMBench, Quicksilver, LULESH, and MiniMDock on an NVidia V100 GPU, and we project their performance on the modern A100 and H100 NVidia GPUs. Effective runs of the latter allow us to compare and validate our projection with actual performance as depicted in Figures 4 to 9. In these Figures, the leftmost blue bar is the average execution time per kernel when running on the V100, the black segment is the predicted performance interval on the target GPU, and the other colored bars are the actual performance measured on the target GPUs. Additionally, the small red line is the mean point of the projection interval. To assess the precision of the prediction, we compute the mean absolute percentage error (MAPE) between the mean prediction, and the actual performance measurement.

5.1 GPU test-bed description

Table 1 summarizes up the characteristics of the different NVidia GPUs used for our experiments.

Table 1: Characteristics of used machines.

GPU	V100	A100-40	A100-80	H100
Compute Capability	7.0	8.0	8.0	9.0
$Perf^{Max}$ (GFLOP/s)	6890	9476	9476	24979
BW_{DRAM}^{Max} (GB/s)	846	1375	1678	1907
BW_{L2}^{Max} (GB/s)	2460	4710	4710	7758
BW_{L1}^{Max} (GB/s)	13963	19492	19492	25330
nvcc version	V12.0.140	V11.6.55	V12.0.140	V12.0.140
CUDA driver version	530.30.02	510.85.02	530.30.02	530.30.02
OS	RHEL 8.8	RHEL 8.8	RHEL 8.8	RHEL 8.8
CPU	2 x 16c Xeon Gold 6226R @ 2.9GHz	2×64c AMD Rome@2.6GHz	2 x 16c Xeon Gold 6226R @ 2.9GHz	2 x 64c Epyc Milan @ 2.8GHz
CPU RAM	512 GiB	256 GiB	512 GiB	512 GiB

5.2 Hydro1D

Hydro1D is a mini-application that solves a hydrodynamic problem. It is mainly a loop composed of 9 consecutive kernels. Each kernel computes a single cell per thread, and there is no reuse of data, such as the DRAM bandwidth is the main limiting factor. We run this application with 50,000,000 cells on the V100 GPU. Figure 4 shows our projection results of the different inner kernels. For all kernels, the prediction intervals are narrow, and predictions are accurate: the percentage error ranges from -5.83% to 4.45%.

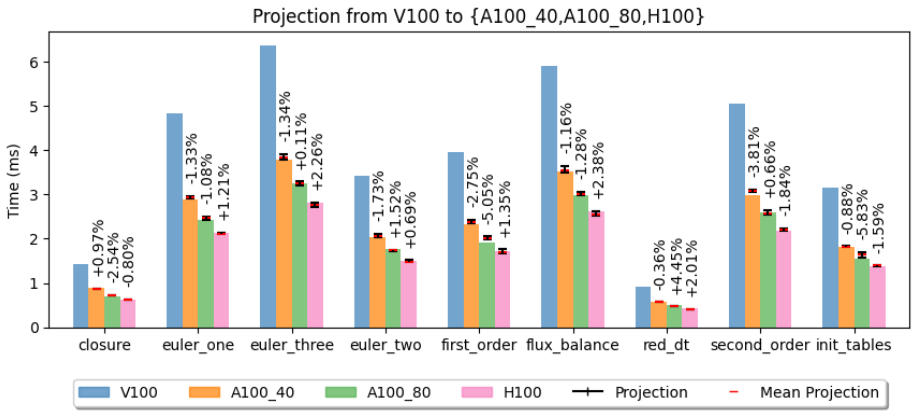


Fig. 4: Performance projection of Hydro1d kernels.

We may specify that we used the Stream Init bench, instead of the Stream Triad one, in order to measure the bandwidth to use for the *init_ tables* kernel as it only writes data, which means a higher bandwidth on the GPU. However, this initialization kernel remains not really relevant in the whole projection performance of Hydro1D.

5.3 UVMBench

UVMBench [9] is a test suite composed of diverse mini-apps, which all have different memory patterns accesses, so that they challenge our memory bandwidth weightings. For this paper, we focus on the polybench 1.0 and KNN mini-apps with parameter configuration listed in Table 2.

We separate the results in two parts: Figure 5 reports the performance prediction results for standard kernels, and Figure 6 reports the performance prediction for kernel that under-use GPUs.

The results for standard kernels show that the performance projection interval are mostly correctly predicted. Some projection intervals are wide (e.g. SYRK, SYR2K, GEMM), which often means that the OI for the DRAM is high, so that the projection is made by taking the maximum performance in

Table 2: UVMBench Parameters.

Benchmark	Parameters	Benchmark	Parameters
KNN	$nb = 16384$	COVAR	$N = 8192$; $M = 2048$
2DConv	$N_{I,J} = 4096$	FDT2D	$N_{X,Y} = 2048$
2MM	$N_{I,J,K,L} = 2048$	GEMM	$N_{I,J,K} = 2048$
3DConv	$N_{I,J} = 1024$; $N_K = 256$	MVT	$N = 32768$
3MM	$N_{I,J,K,L,M} = 4096$	SYRK	$N = M = 1024$
ATAX	$N_{X,Y} = 32768$	SR2K	$N = M = 1024$
COOR	$N = 8192$; $M = 2048$	GRAMMSC	$N = 32768$; $M = 131072$

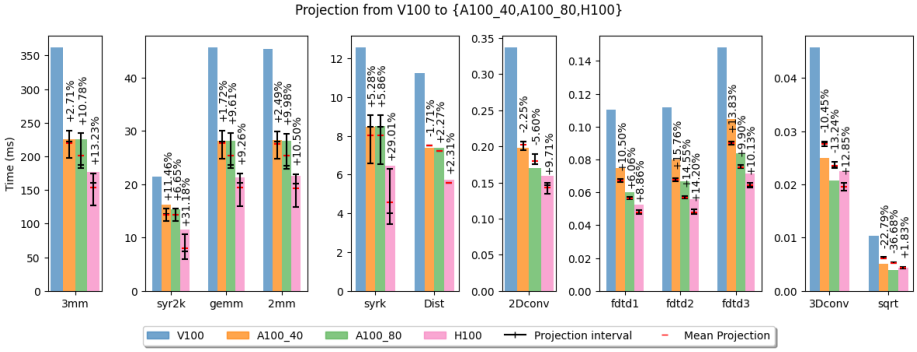


Fig. 5: Performance projection of UVMBench for intensive kernels.

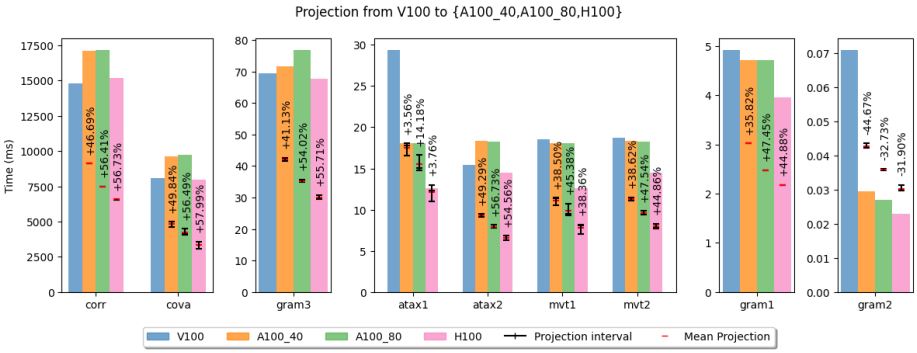


Fig. 6: Performance projection of UVMBench for kernels that under-use GPUs.

Equation (9), which is quite different from one GPU generation to another, see Table 1.

Figure 6 shows that when under-using GPUs, several kernels (e.g. corr, or gram3) do not benefit from running on powerful GPUs due to their lack of parallelism. Since our performance projection does not take this into account, the prediction are inaccurate.

5.4 Quicksilver

Quicksilver [18] is a proxy application that solves a Monte-Carlo particle transport problem. The GPU version is made with a unique kernel and works with unified memory, which may imply memory latency. The application is composed of loops in which batches of particles are computed in the kernel one after the other, such that at the beginning of an iteration, a maximum amount of particles are computed during a single kernel, while at the end only the remaining particles are computed. Our testing case is the "Coral2_P1_1" problem with nParticles = 1000000.

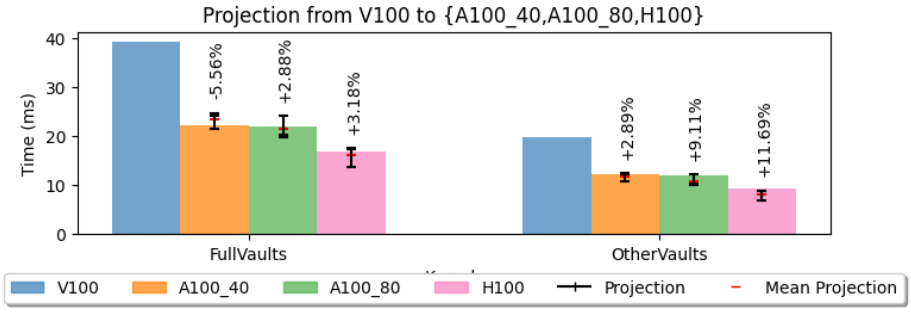


Fig. 7: Performance projection of Quicksilver.

Figure 7 presents our projection from the V100 GPU to the other GPUs on two different key moments of the kernel. The "FullVaults" case appears at the beginning of the time step when lots of particles are computed during the kernel, whereas "OtherVaults" is for the other cases. Cases with very few particles (using less than 30 blocks of threads) are excluded. It has to be noted that Quicksilver has a very low number of active threads per warp instructions: about 6 over 32, which makes it a very poor performing kernel. Despite not having taken into account this particular metric for weighting the bandwidths used for the projection, our prediction remains correct with a MAPE of 5.9 %.

5.5 LULESH

LULESH [10] is a mini-application that models 3D Lagrangian hydrodynamics. It is composed of a typical loop that iterates on a kernel that computes a time

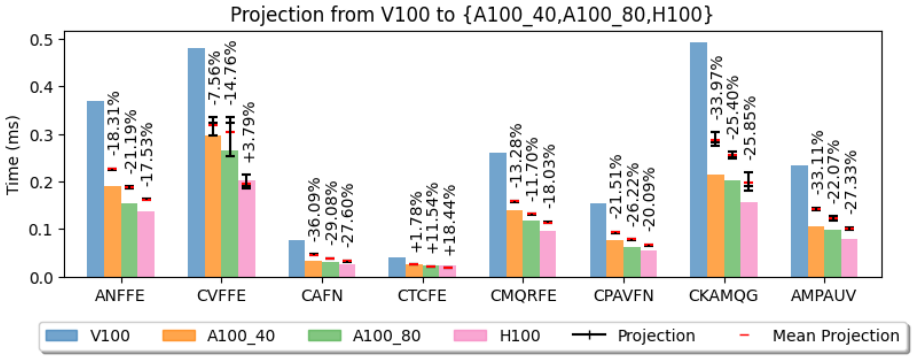


Fig. 8: Performance projection of LULESH.

step in which each thread computes a single mesh. The DRAM bandwidth is here the main restricting factor.

As depicted in Figure 8, the projected execution time is often higher than the actual measured execution time on the target GPUs. The V100 seems to have a different behaviour than the other GPUs with these kernels, probably due to it smaller L2 cache size. Indeed, it implies a lower memory reuse for data that may be in L2 during runs on others GPU architecture. Even if we correctly calculated that the OI at level L1 is higher on V100, because of more local memory operations made in the other GPUs, the OI at DRAM level is far lower on V100 due to more memory transfers between the DRAM and the L2 cache according to the profiler. If we had taken in consideration the cache size, then the prediction would have been correct, since the DRAM bandwidth is the main restraining factor for these kernels. Overall, the MAPE is about 20.26 %.

5.6 MiniMDock

MiniMDock [19] is a molecular docking mini-application for which we use the default input 7cpa ligand and 100 LGA runs. This application makes use of the shared memory for its main kernel, but also not all threads of a warp are used during each instruction.

Figure 9 presents the projection of three kernels : two initialization kernels, CALC_INITPOP and GEN_AND_EVAL_NEWPOPS, and the computation kernel one PERFORM_LS. The measured performance on the target GPUs are in the predicted intervals, except when projecting performance to the H100 GPU.

While bandwidths are modeled assuming that all warp threads are active, the use of a performance analysis tool shows that, on average, only 26 threads over 32 per instruction are here active, which leads to inaccurate projections.

The projection error for both the A100s is virtually the same because the DRAM OI is high (above 1000 FLOP/Byte), which means that the DRAM bandwidth, which is the main difference between the two A100s, has virtually no influence in our performance model, hence the same performance projected.

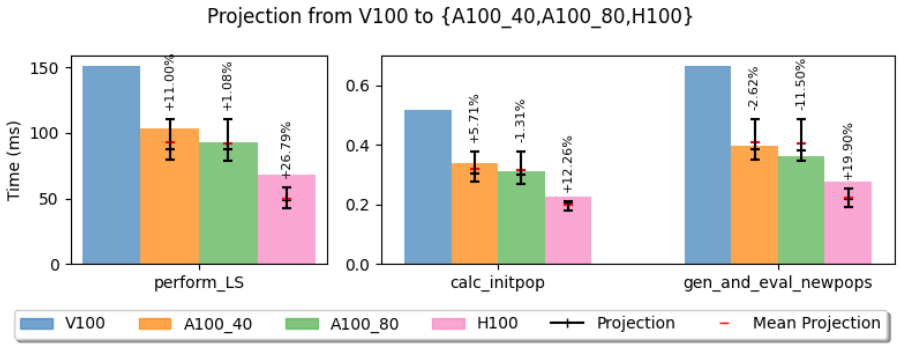


Fig. 9: Performance projection of MiniMDock.

6 Related Work

Several approaches have been used to project the performance from one architecture to another, including simulation, statistical model with machine learning and analytical model.

Simulators such as Gem5 [5] can accurately estimate the performance of an application on a target computer architecture. However, such simulation consume extensive resources and they are not viable for large applications. In this context, less accurate but faster methods such as performance projection models become interesting. For example, Domke et al. [7] propose a method to easily and quickly calculate the upper bound in performance improvements when changing cache components. As a result, modeling the behavior of a CPU only takes a few hours, instead of months with cycles-accurate simulators.

Simulators like GPGPU-SIM [3] and its extension Accel-Sim [11] reproduce the behaviour of a GPU by analyzing the instructions of a kernel obtained during a previous execution on a GPU, and on characteristics of the targeted GPU. Despite their precision, their large overhead and large size of traces needed to run such simulations make them impractical for real-life applications.

Instead of simulating the whole GPU and the kernel, which is resource consuming, one can reduce the cost of performance projection by using a performance model. Micro-benchmarks first characterize the GPU. Then, metrics are collected by analyzing the kernel, e.g. by executing it. Depending on the number of parameters to collect and to use, one can construct either machine learning (ML) models [4] to handle many parameters at a cost of a time-consuming data collection and a training for each studied GPU, or analytical models [20], which relies on simplified assumptions where kernels are classified depending on their main restraining factor. Moreover, CPU performance prediction are good work starting point. In [2], a machine learning model analyzes cross CPU-GPU applications first with only a CPU, before projecting their performance on a GPU. While in [8, 13], the roofline model has been used for projecting an application performance on CPU architectures, based on its performance on a source CPU.

7 Conclusion

Throughout this paper, we present a methodology to get a performance interval of a given kernel for a target GPU by first analyzing the performance of the kernel on a source GPU. We base our performance prediction on a roofline model with multiple ceilings that are specific to the kernel. These ceilings characterize the behaviour of a kernel depending on its instruction mix, cache hits and warp efficiency. Moreover, we estimate the operational intensity of the kernel on the target GPU by comparing the assembly instructions of its executable with traces obtained on the source GPU. We then used a set of modern NVIDIA GPUs (V100, A100 40 & 80 GB, and H100) to evaluate our methodology on several mini-apps (Hydro1d, UVMBench, Quicksilver, LULESH and MiniMDock). This methodology is quite fast to run, since it only requires to profile the kernel of interest with a sample run on a source GPU to project its performance on other GPUs. The evaluation shows that we correctly project the performance from the V100 GPU to the other GPUs on our studied kernels.

In the future, we plan to extend this methodology to predict the performance of the whole application, and not only of its kernels. Moreover, modifications may be needed for predicting on AMD and Intel GPUs and also for exploring the impact of the unified memory.

Acknowledgments. We thank the University of Oregon and the OACISS team for the use of their machines.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abdelkhalik, H., et al.: Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis (2022)
2. Ardalani, N., et al.: Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In: Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (2015)
3. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software (2009)
4. Benatia, A., Ji, W., Wang, Y., Shi, F.: Machine Learning Approach for the Predicting Performance of SpMV on GPU. In: IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS) (2016)
5. Binkert, N., et al.: The gem5 simulator. ACM SIGARCH Computer Architecture News (2011)
6. Ding, N., Awan, M., Williams, S.: Instruction Roofline: An insightful visual performance model for GPUs. Concurrency and Computation: Practice and Experience (2022)
7. Domke, J., et al.: At the locus of performance: Quantifying the effects of copious 3D-stacked cache on HPC workloads. ACM Transactions on Architecture and Code Optimization **20**(4), 1–26 (2023)

8. Gavoille, C., et al.: Relative Performance Projection on Arm Architectures. In: Euro-Par 2022: Parallel Processing. Lecture Notes in Computer Science
9. Gu, Y., Wu, W., Li, Y., Chen, L.: UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs (2020)
10. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Tech. rep. (2013)
11. Khairy, M., Shen, Z., Aamodt, T.M., Rogers, T.G.: Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In: ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA) (2020)
12. Konstantinidis, E., Cotronis, Y.: A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distributed Computing (2017)
13. Kwack, J., et al.: Roofline analysis with Cray performance analysis tools (CrayPat) and roofline-based performance projections for a future architecture. Concurrency and Computation: Practice and Experience (2019)
14. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (1995)
15. NVIDIA: CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
16. NVIDIA: Nvidia Nsight Compute, https://docs.nvidia.com/nsight-compute/NsightCompute/index.html
17. Petitet, A., et al.: HPL – a Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers (2008)
18. Richards, D., Brantley, P., Dawson, S., Mckenley, S., O'Brien, M.: Quicksilver, version 00 (2016), https://www.osti.gov/biblio/1313660
19. Thavappiragasam, M., et al.: Performance portability of molecular docking miniapp on leadership computing platforms. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (2020)
20. Wang, Q., Chu, X.: GPGPU Performance Estimation With Core and Memory Frequency Scaling. IEEE Transactions on Parallel and Distributed Systems (2020)
21. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM (2009)
22. Yang, C., Kurth, T., Williams, S.: Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. Concurrency and Computation: Practice and Experience (2020)
23. Yang, C., Wang, Y., Kurth, T., Farrell, S., Williams, S.: Hierarchical roofline performance analysis for deep learning applications. In: Intelligent Computing: Proceedings of the 2021 Computing Conference, Volume 2. pp. 473–491 (2021)
24. Yang, C., et al.: An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (2018)