HPC on a budget: on the CPU's impact on dense linear algebra computation with GPUs

Lucas Barros de Assis[1], **Lucas Mello Schnorr**[1]

[1] Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{lbassis, schnorr}@inf.ufrgs.br

Abstract. *While GPU-equipped nodes dominate compute-intensive operations in high-performance computing, the role of host processors in overall system performance remains underexplored, particularly for hardware procurement decisions in resource-constrained environments. This study investigates the influence of the CPU microarchitecture on application performance using an identical modern GPU. We compare two systems with the same NVIDIA RTX4090 GPU but different CPUs: a 2016 Intel Xeon E5-2620v4 and a 2023 Intel Core i9-14900KF. We assess CPU impact on compute-bound workloads using dense Cholesky and LU factorizations from the Chameleon library assisted by the StarPU runtime system. Our findings demonstrate that: (1) CPU influence is negligible with modern GPU accelerators, even if some operations lack GPU implementations; (2) CPU-handled operations are sufficiently small to attenuate performance differences between processor generations; and (3) modern GPUs perform effectively in legacy hardware with minimal penalties. These results suggest selective GPU upgrades offer cost-effective performance improvements without complete system overhauls, providing valuable insights for academic and research institution procurement strategies.*

1. Introduction

Heterogeneous architectures, with GPU-equipped nodes, are commonplace in today's clusters and are widely adopted in High-Performance Computing (HPC) environments [Dongarra and Keyes 2024]. Thanks to these accelerators' massive throughput, way beyond that attainable by general CPU cores, all vectorized HPC applications profit essentialy from them. GPU accelerators have become essential for dense linear algebra operations, such as LU and Cholesky factorizations [Tomov et al. 2010], but extend to many other applications, including fast matrix multiplication kernels supporting training and inference for artificial intelligence applications [Talib et al. 2021]. For example, dense linear algebra libraries such as Chameleon [Agullo et al. 2010] provide kernels that exploit the performance these architectures can deliver. Consequently, there is a widespread focus on GPUs for compute-bound applications in HPC. Nonetheless, the CPU still has its place, as even if all the computation happens on the accelerators, at the very least, communication and I/O operations still pass through the CPU.

The reality of heterogeneous computing involves complex interactions, including data movement between the host and GPU, the PCIe bandwidth limitations, and numerous other system-level considerations that impact overall throughput. From the software perspective, the inherent task scheduling complexity

in effectively combining PCIe-based CPU and GPU power has become an issue to be studied [Tan et al. 2021], primarily addressed through task-oriented programming paradigms following the Sequential Task Flow (STF) [Pei et al. 2022] paradigm. Several sophisticated runtime systems have emerged to tackle these challenges, including StarPU [Augonnet et al. 2009], PaRSEC [Bosilca et al. 2013], Specx [Cardosi and Bramas 2025], TaskTorrent [Cambier et al. 2020], OpenMP Target Task [Valero-Lara et al. 2021], and CHAMELEON [Klinkenberg et al. 2020]. They employ clever performance modeling techniques that account for resource (CPU and GPU) capabilities and computation and host-GPU communication overhead. The goal is to make good decisions about where to schedule ready tasks. This sophisticated orchestration reveals that the CPU's role extends far beyond simple data management, as it actively participates in the critical scheduling decisions that can dramatically influence the efficiency of GPU-accelerated computations.

System administrators regularly synchronize GPU-enabled host configurations with the latest GPU release, upgrading entire systems to match new GPU generations. However, GPU releases are more frequent nowadays than in the past, and one wonders if it is necessary to update the whole system infrastructure to host a more modern GPU, particularly given the substantial costs involved in complete system refreshes. One needs, therefore, to understand how a modern GPU behaves when deployed in older hardware environments, including legacy CPUs, older PCIe generations, and previous-generation system architectures, specifically from the performance perspective. Such understanding might affect hardware procurement decisions, especially in budget-constrained environments where updating only the GPU might be a viable and cost-effective path.

This article investigates the influence of different CPUs' microarchitectures on application performance when using recent identical GPUs. We leverage the tiled and task-based dense Cholesky and LU factorization of the Chameleon library [Agullo et al. 2010] running on top of the StarPU runtime [Augonnet et al. 2009] as the primary benchmark. We select it because of its compute-bound characteristic and presence in numerous HPC applications. We carried out the comparison with an NVIDIA RTX4090 GPU, and two CPUs: a 2016 Intel Xeon E5-2620v4, and a 2023 Intel Core i9-14900KF processor. The main contributions of this work include: **(1)** the CPU influence is negligible when working with such accelerator, even if some of the operations have no implementation for GPU; **(2)** the amount of CPU operations is small, attenuating performance differences; and **(3)** modern GPUs in old hardware bring acceptable performance penalties subject to the configurations used in our experimentation. Combined, these results points to selective GPU as a cost-effective performance improvements without complete system overhauls.

Section 2 presents core concepts regarding task-based applications and tracing. Section 3 highlights related work and a discussion motivating our work. Section 4 discusses the experimental methodology used in this work, with details about the applications, tools, and systems used in the experiments. Lastly, Section 5 shows and discusses the experimental results, while Section 6 concludes this work. A publicly available companion[1] includes the paper's data and visualization code to enable experimental reproducibility tight to this work.

[1] `https://github.com/lbassis/cpu_impact_companion`

2. Background

Our experimental work combines the StarPU runtime system and the Chameleon framework. A summary of their utilisation comes in the following subsections.

2.1. StarPU

StarPU [**?**] is a library for task-based programming on heterogeneous architectures. The task-based paradigm allows its users to write code as *tasks*, which are built by combining small routines and their data dependencies. When working with parallel dense linear algebra, these tasks correspond to subroutines that work independently over different parts of the input data. With these tasks, StarPU creates, at runtime, a direct acyclic graph describing the order in which these tasks can be executed in parallel on the available hardware, if the dependencies allow it. The StarPU library also provides a runtime scheduler that distributes the available tasks on the hardware. By registering the time taken to execute its tasks, *StarPU* can create performance models that describe the behavior of the employed cores. StarPU's scheduling policies (such as the `dmdas` adopted in this work) use these models to estimate task completion time.

2.2. Chameleon

Chameleon [Agullo et al. 2010] is a framework that implements dense linear algebra routines for heterogeneous architectures in C, supporting real and complex arithmetic in single and double precision. Since its code is written for heterogeneous architectures, when executed with a runtime system such as StarPU, the Chameleon framework provides specific implementations aimed at the specificities of the adopted underlying hardware. For example, tasks might use CUDA to run on GPU devices, or regular C code to run on CPU cores. Other than the dense linear algebra operations, the framework also provides test binaries to assert its algorithms' correctness and extract performance metrics such as the operations *GFLOPS* and the execution time. This study employs these test binaries to measure the performance when comparing the two different setups available.

3. Related Work

Research specifically evaluating modern GPUs deployed in older CPU configurations remains notably scarce in the literature. Most GPU performance studies deliberately pair accelerators with contemporary CPUs to avoid introducing bottlenecks, leaving system administrators without adequate guidance for evaluating selective hardware upgrades in resource-constrained environments.

A relatively recent work [Li et al. 2019] evaluates how modern GPU interconnect technologies impact the performance of multi-GPU applications. The paper provides a comprehensive evaluation of five key interconnect types (PCIe, NVLink, NV-SLI, NVSwitch, and GPUDirect) across high-end server and supercomputing platforms. Other similar and more recent investigations focus on memory bandwidth [Mishra et al. 2024], and GPU scalability [Tan et al. 2021]. Another investigaton [Xiao et al. 2018] considers that CPU influence is effectively underevaluated for Deep Learning (DL) jobs in HPC clusters, providing valuable insights into CPU resource allocation. The work focuses on cluster-level job scheduling and resource allocation across multiple DL jobs running simultaneously on a GPU cluster, treating individual applications as black boxes that compete for CPU resources. In contrast, our study examines the intra-application performance

impact of host configuration within a single dense linear algebra application. StarPU-specific investigations on the matter of questioning if an host upgrade is necessary when a new GPU appears also remain scarse. Recently, a novel algorithm [Gonthier et al. 2022] tackles the problem of memory-aware scheduling of tasks sharing data on multiple GPUs. More generally, many runtime systems employ scheduling heuristics that considers host-GPU transfer cost [**?**], and partitioning schemes [Navarro et al. 2014], pointing to the importance of alignment between host and GPU configuration.

Although the PCIe and communication hiding techniques play a clear role on the overall GPU performance, the analysis of the related work points to a general understanding that a powerful GPU requires an adequate host configuration. However, it is clear by the scarcity of investigations on this sense that one can still wonder if modern runtime systems are still capable to exploit maximum GPU performance in an relatively old host configuration. As most of the previous work are purely GPU-centric, our work differs because we focus on the influence of the host configuration when employing StarPU to schedule tasks in CPU cores and GPU for a compute-bound workload.

4. Methods and Materials

We first describe the hardware and software configuration adopted in this work. We then detail our foundational performance-impacting studies that drive our methodological effort to carry out the main goal described in this article. These studies include: (1) runtime system calibration, (2) the effect of heterogeneous cores modeling in the i9 processor; (3) our baseline CPU-only measurements to demonstrate the performance gap between the studied processors; (4) the evaluation whether we should dedicate or not a single CPU core to handle exclusively CUDA operations. These studies then allowed us to define the parameters that allow *StarPU* to extract all the performance it can from our experiments, described at the end of this section.

4.1. Hardware & Software configuration

We employ four nodes of the `tupi` partition of the PCAD environment at INF/UFRGS in our experiments. Table 1 specifies the hardware for each of them: tupi[1-2] shares the same configuration, equipped with a 8-core Xeon Broadwell processor (2.1Ghz); tupi[3-4] also shares the same configuration, being equipped with a modern i9 Skylake processor with 8 P-cores (3.2Ghz), and 16 E-cores (2.4Ghz). We chose to test in these different hardware specifically because of the different CPU models. All four nodes have each a GeForce RTX4090 connected in the host throught the PCIe 3.0 (Xeon) and 4.0 (i9). We adopt the Spack package manager [Gamblin et al. 2015] to control our software stack and make reproducible the software environment in a Debian 12 host. We set Spack to compile all software for each CPU's own microarchitecture: Broadwell for the Xeon, and Skylake for the i9. We used the StarPU runtime system on commit `146ce9d8` combined with the Chameleon library on commit `3e958439`. We employ the `dmdas` scheduling policy of StarPU. It schedules tasks where their termination time will be minimal according to performance models created on previous executions, but also considering the data transfer time, the priority and the state (whether ready or not) of the available tasks [StarPU Project 2015]. The nodes had CUDA 12.3 and the NVIDIA driver version 545.23.08 for `tupi[1-2]` and CUDA 12.4 and the the NVIDIA driver version 550.54.15 for `tupi[3-4]`. We control the typical experimental parameters such as CPU frequency

and governor, GPU frequency, Hyper-threading and Turboboost to reduce variability in our experiments. For instance, we kept disabled the Hyper-threading technology since it is known to degrade the performance on compute-bound applications, which is the case for the selected parallel dense linear algebra applications.

Table 1. Hardware specification of the machines used for the experiments

Name	CPU	RAM	PCI	GPU
`tupi[1-2]`	Xeon E5-2620v4 2.1Ghz	256GB DDR4	PCIe3.0 x16	RTX4090
`tupi[3-4]`	Core i9-14900KF 3.2Ghz	128GB DDR5	PCIe4.0 x16	RTX4090

4.2. Foundational performance-impacting studies

Combining Chameleon's implementation for dense linear algebra routines with StarPU's runtime system, it is possible to run a series of operations on the available hardware to properly compare their performances. However, to obtain all the computing power these tools can provide, one must first tune their parameters to the available machines. Our studies lead then to a proper design of experiments. They are detailed as follows.

Runtime system calibration: tuning the StarPU models for our experiments. Since the adopted `dmdas` scheduling policy requires a performance model for each kernel, it is important to take their accuracy into account. Every time a kernel is executed, the runtime system stores its execution time and the hardware where it was ran, and then updates an average execution time for this specific configuration. StarPU considers that a performance model is calibrated when the difference between the last execution time and its configuration previous average is under a threshold value, set by the `STARPU_HISTORY_MAX_ERROR` environment variable. From that point and on, StarPU considers models to be calibrated and no longer update them unless another measurement surpasses the threshold cited beforehand. By default, StarPU creates a single model for each CPU; however, it also implements an option to consider each core separately in order to explore heterogeneous CPU cores.

We demonstrate the calibration effort when computing a single precision LU factorization on a matrix of order 32k. All of the kernels involved (*sgemm*, *splgsy*, *strsm* and *sgetrf_nopiv*) were fully calibrated according to the default threshold of 50%. Investigating further this issue, we set different threshold values, ranging from 10% to 50%, to check its influence on the resulting models. Figure 1 depicts the resulting average times (in the Y-axis) as a function of each model (the facets) and calibration thresholds (in the X-axis), both for the Xeon (blue color) and i9 (red color) CPUs. Despite our concerns to have a fitting performance model for each task with a higher calibration threshold, the results demonstrate the impact of this parameter is negligible. The Figure 1 also shows that the *sgemm* and *strsm* create three distinguishible clusters: one for the Xeon cores and two for the i9 performance and efficient cores. This behavior is expected as the i9 has P and E-cores with different compute capabilities. For example, considering the `sgemm` kernel, the i9's E-cores demonstrate an average execution time for the `sgemm` operation of about 60000us, while the P-cores of this same processor runs the same operation at about a third, around 20000us. What is surprising is that the models for P and E-cores do

not differ so much when computing the *splgsy* kernel. Although ommited from this figure to allow for a proper Y-scale, all GPU models displayed a similar behavior.

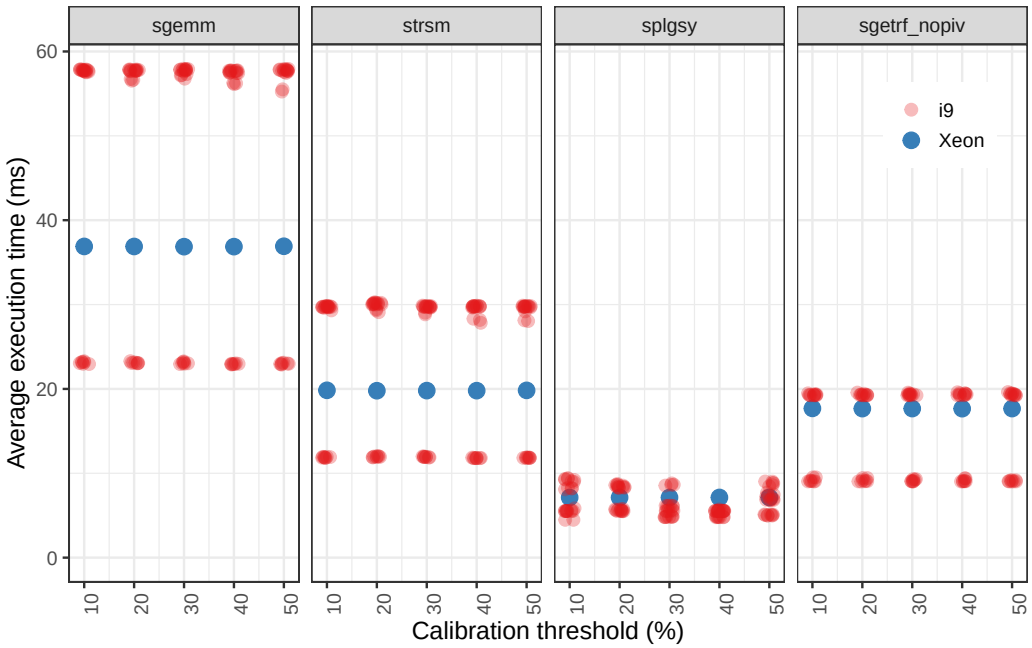

Figure 1. Average execution time as a function of calibration thresholds (LU).

Per-core performance model in the i9 processor: effects on makespan. Since we employ the i9 processor with its heterogeneous CPU cores, it is necessary to compare application performance when dealing with the StarPU's capability to create a per-core performance model. Before illustrating the effects on the makespan on the usage of per-core performance models, we carried out a traced execution to collect the duration and core placement for every single `dgemm` operation. By using these execution's traces, Figure 2 presents a recreation of each core's model, depicting the `dgemm` performance (average execution time in ms) as a function of the CPU core. In the i9 processor, core identifiers from CPU0 to CPU6 are P-cores, thereby with a smaller average execution time (≈50ms), while cores from CPU7 to CPU22 are E-cores, with a larger average execution time (≈115ms) for the `dgemm` operation. Note that one P-core is absent because it was dedicated to handle CUDA operations. This result clearly confirms that there is indeed a significant difference between the P-cores and the E-cores performance, matching StarPU calibrated performance models. One should then expect these per-core models would provide a better overall application makespan.

Using the same traces, we then compare the overall makespan execution time of the whole application to assess the impact of per-core performance models. Figure 3 shows the average execution time (in the Y-axis) as a function of the block size (in the X-axis), the two applications (facets), and the usage an homogeneous single model (blue color) and heterogeneous per-core models (red color). Even though the models are completely different for each configuration, as we observed previously, the resulting makespans when comparing the two colors for each block size are very similar for block sizes 512 and 1024, with a difference for the larger block size (2048), where the per-core models provide indeed a small advantage. However, when comparing with the

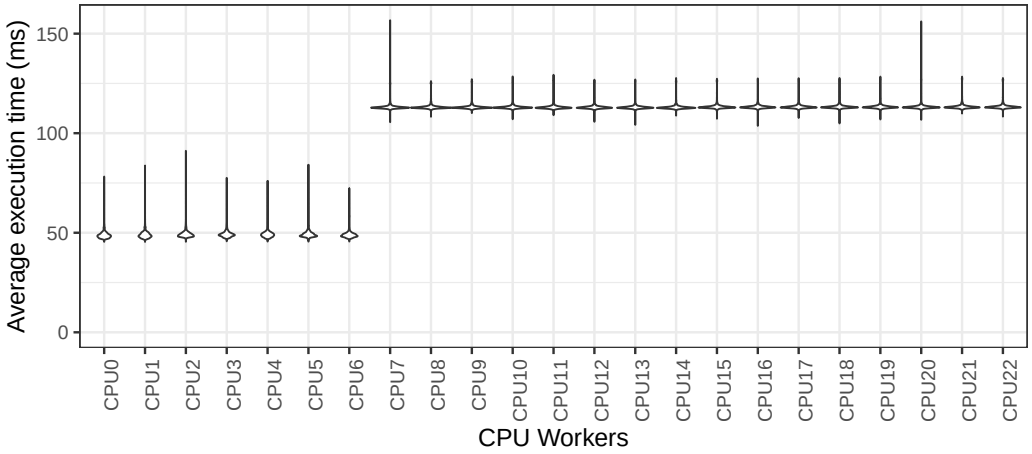

Figure 2. *dgemm*'s average execution time on each core.

performance with different block sizes, it is visible that the one who profits from the heterogeneous models has the worst performance between the studied values.

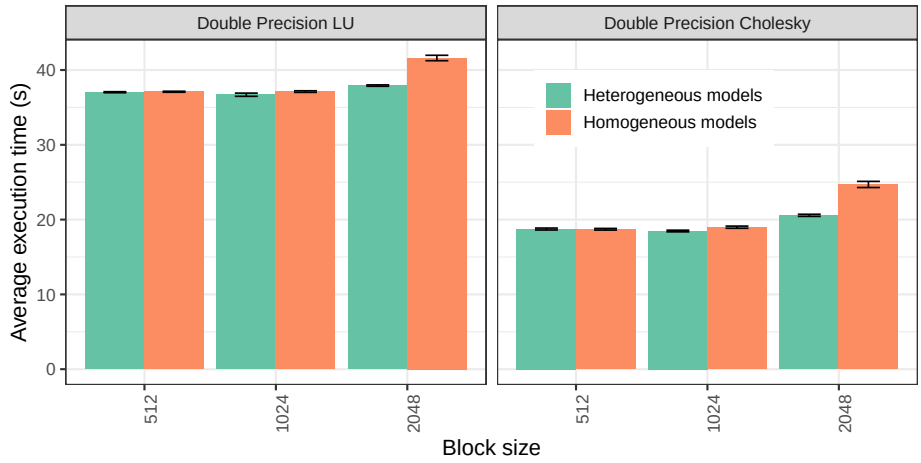

Figure 3. Comparing homogeneous and heterogeneous models.

Despite the fact that we indeed observe different performances for `dgemm` when comparing P and E-cores of the i9 processor (see Figure 2), we were unable to verify an actual effect on overall application makespan (see Figure 3) for any block size. Regardless of our efforts, we were unable to determine precisely the reason why the per-core models cannot provide a better overall makespan for smaller block sizes. We leave this investigation as future work, with the understanding that performance is dominated more by scheduling decisions and load balancing than by per-core performance. Because of such reasons, we decided to adopt homogeneous performance models for both i9 and Xeon processors.

CPU performance comparison between i9 and Xeon: baseline definition. We determine a baseline for this study, effectively quantifying the CPU performance differences between i9 and Xeon. Figure 4 shows the average execution time (in the Y-axis), along with the standard deviation metric, as a function of the application (facets) and different block sizes (in the X-axis) and processors (colors). As expected, largely because of

the higher CPU base frequency, the i9 is significantly faster than the Xeon. The application execution takes aproximately half of the time to finish in the i9 compared against the Xeon. Regarding the block size, the results demonstrate that for both of the processors, the optimal block size is somewhere around 1000 for a single precision input. This foundational results indicates clearly that there is an enourmous performance difference between the two processors when running without a GPU accelerator.

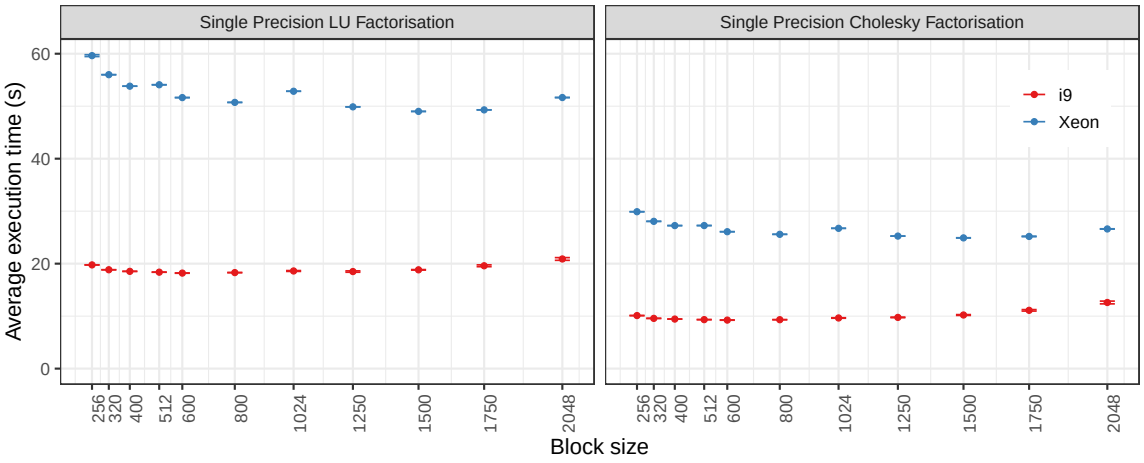

Figure 4. CPU-only execution on the i9 and Xeon processors.

Dedicated CPU-core to handle CUDA operations: assess performance. Finally, when working with CUDA, StarPU allows the user to determine which CPU core will perform the CUDA operations (i.e. the data transfers from the CPU to the GPU) and if it will work exclusively or not on these instructions. Without dedicating a CPU-core for CUDA operations, the same core is subject to receive its share of tasks to compute. This possibility in the StarPU configuration added a new variation to our GPU experiments, either to dedicate a CPU core for CUDA (exclusive thread) or not. Figure 5 depicts the results on this matter. The figure shows the average execution time (Y-axis) as a function of the application (top and bottow facet rows), the block size (X-axis) for the Xeon processor. The color differentiates whether there is a CUDA exclusive thread (red color) or not (blue). The results indicate that dedicating a CPU core to manage a CUDA resource improves performance.

4.3. The final Design of Experiments (DoE)

Our design of experiments have the following factors and levels: (1) application, LU and Cholesky factorization; (2) floating-point precision, single and double; (3) input size, square matrices of orders 32768 and 100000; (4) a diverse set of block sizes; (5) the amount of CPU and GPU workers, as one choice for each type of machine (Xeon with 8-cores, and i9 with 8 P-cores and 16 E-cores). All the experiments dedicated one thread exclusively to the CUDA operations. With the Chameleon performance-oriented binaries, we repeat each experimental combination six times on a random fixed order to scatter the execution of same configuration runs. We finally remove the worst execution to mitigate the known effects of StarPU's calibration mechanism.

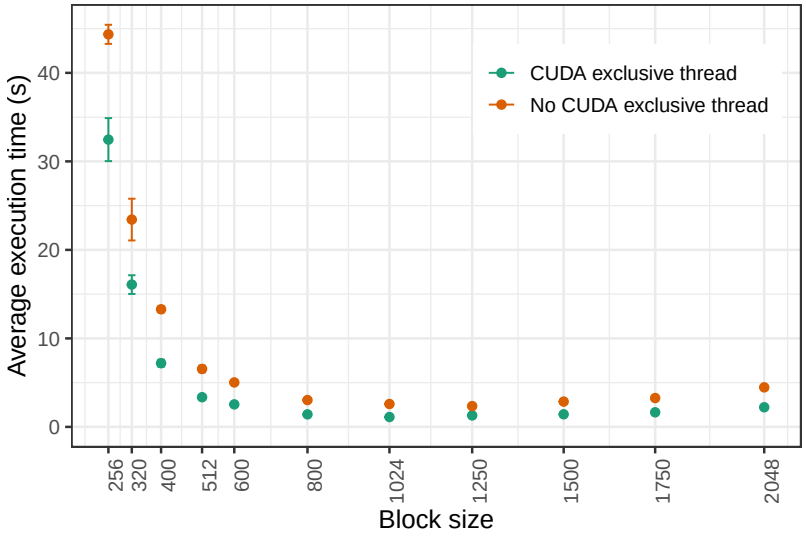

Figure 5. Performance as a function of an exclusive CPU-core for CUDA calls.

5. Results

When comparing the optimal StarPU's configuration, Figure 6 shows that the difference between the two CPU's performance shrinks as we approach Chameleon's optimal block size. This result can also be reinforced by checking the performance models: looking at the single precision LU factorization on the Xeon CPU, for example, even if the *splgsy* and the *sgetrf* kernels are only implemented on the CPU, ≈90% of the kernels are executed on the GPU; on the *sgemm*, the most executed kernel, the GPU is responsible for ≈99% of the computations. These results show that as long as the CPU is fast enough to control the data transfers and the task's scheduling, the GPU takes care of the largest part of the computation work. Furthermore, one can see the variability growth on the i9 when its block size surpasses its optimal size. Furthermore, looking past the optimal block size on the i9, one can see that the performance gap between the P-cores and E-cores show up as a progressive variability.

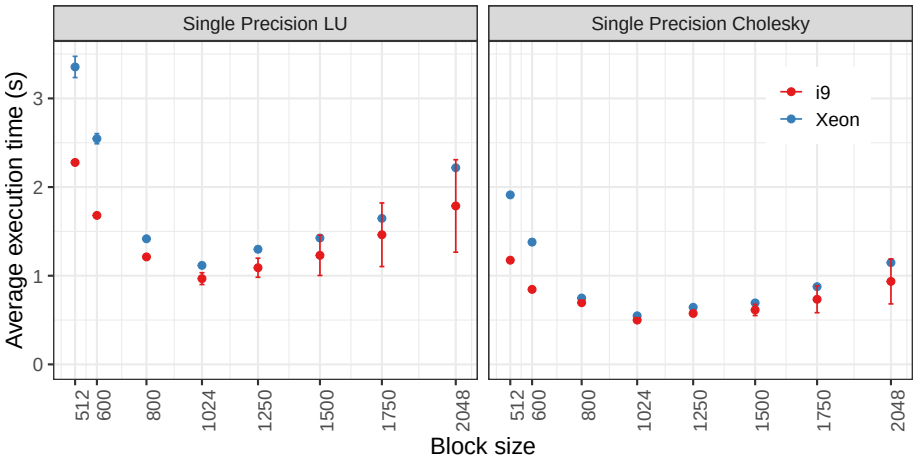

Figure 6. Experiments employing both CPU and GPU.

Figure 7 shows a closer look at the execution times on single precision for different block sizes. Even though the i9 performs better on every configuration, we would

like to discuss their behavior with different block sizes: this figure shows that setting an inadequate value for this parameter is enough for the i9 to perform worse than the Xeon: for instance, the Xeon with blocks of size 1024 outperforms the i9 with blocks of size 800, highlighting the importance of a proper configuration for each individual CPU.

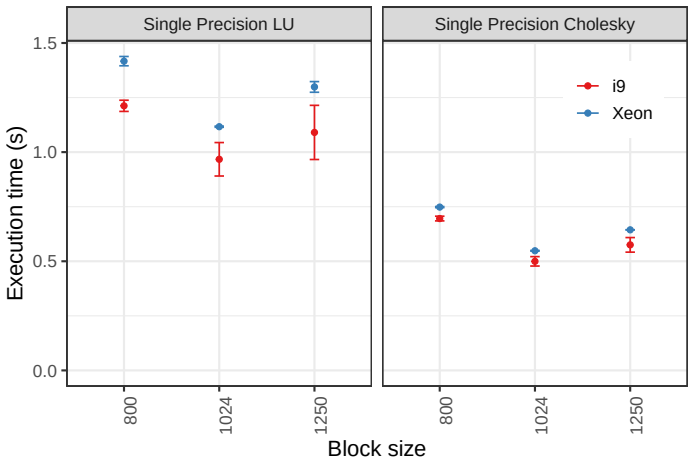

Figure 7. Closer look at the impact of the block size between CPUs.

Even if the machines performance seems to converge, the aforementioned experiments had smaller matrices which did not fully explore the GPU's capacity. To maximize GPU performance, we also compare matrices of order 100k, using almost 90% of the GeForce RTX 4090 available memory. Figure 8 indicates that as we approach the optimal block size the gap in performance between processors is attenuated, to the point of becoming statistically equal in some cases.

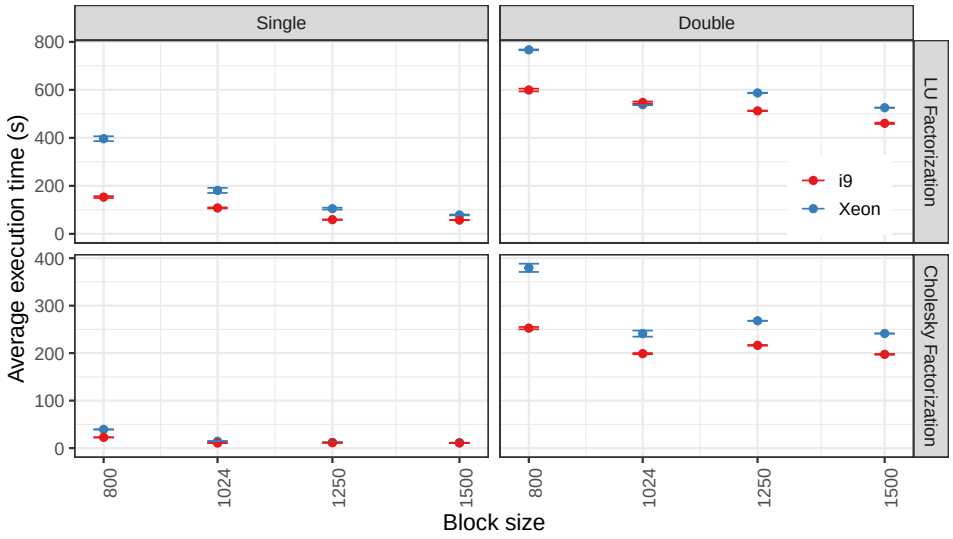

Figure 8. Experiments with matrices of order 100k to occupy as much of the GPU's memory as possible.

The fastest configurations found in this study were obtained by having a CUDA exclusive thread with block sizes of 2048 (for single precision) and 1024 (for double precision), with the i9 performing ≈20% faster than the Xeon, except for the double precision Cholesky factorization, when the gap dropped to ≈13%. Additionally, the smallest

difference with the same configuration was found with matrices of order 100k on the single precision Cholesky factorization. In this case, while the Xeon took 11.3 seconds, the i9 finished its computation in 10.8 seconds, only 5% faster.

6. Conclusion

In this study we used different machines to track the impact of their CPUs when they are equipped with the same GPUs which are known to perform the vast majority of the computations. Our results suggest that, under a cost-benefit perspective, there is no need to upgrade whole machines; as long as the CPU can keep up with its communication and scheduling tasks, the GPU delivers such a significant performance that in can exceed a CPU without its optimal parameters. We also explored StarPU's parameters to a certain extent. The results obtained show that employing a core to handle exclusively CUDA operations improves performance. At the same time, when dealing with the i9's hybrid architecture, which combines P-cores and E-cores of different capacities, we saw that the creation of specific models for each core is not enough to profit from this microarchitecture's novelty. From all the analysis made, two questions remain to be answered on future work: how was the Xeon able to outperform the i9 on some specific cases and how can we profit from the core-specific models provided by StarPU.

Acknowledgments. We would like to thank the PCAD at INF/UFRGS for making infrastructure and hardware used in the experiments available. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, the FAPERGS (16/354-8, 16/348-8), Petrobras (2020/00182-5).

References

Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., and Tomov, S. (2010). Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, volume 2. Morgan Kaufmann.

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer.

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., and Dongarra, J. J. (2013). Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45.

Cambier, L., Qian, Y., and Darve, E. (2020). Tasktorrent: a lightweight distributed task-based runtime system in c++. In *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+ X (PAW-ATM)*, pages 16–26. IEEE.

Cardosi, P. and Bramas, B. (2025). Specx: a c++ task-based runtime system for heterogeneous distributed architectures. *PeerJ Computer Science*, 11:e2966.

Dongarra, J. and Keyes, D. (2024). The co-evolution of computational physics and high-performance computing. *Nature Reviews Physics*, 6(10):621–627.

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., and Futral, S. (2015). The spack package manager: bringing order to hpc software chaos. In *Intl. Conf. for High Perf. Comp., Networking, Storage and Analysis*.

Gonthier, M., Marchal, L., and Thibault, S. (2022). Memory-aware scheduling of tasks sharing data on multiple gpus with dynamic runtime systems. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 694–704.

Klinkenberg, J., Samfass, P., Bader, M., Terboven, C., and Müller, M. S. (2020). Chameleon: reactive load balancing for hybrid mpi+ openmp task-parallel applications. *Journal of Parallel and Distributed Computing*, 138:55–64.

Li, A., Song, S. L., Chen, J., Li, J., Liu, X., Tallent, N. R., and Barker, K. J. (2019). Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110.

Mishra, S., Chakravorty, D. K., Perez, L. M., Dang, F., Liu, H., and Witherden, F. D. (2024). Impact of memory bandwidth on the performance of accelerators. In *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, PEARC '24, New York, NY, USA. ACM.

Navarro, A., Vilches, A., Corbera, F., and Asenjo, R. (2014). Strategies for maximizing utilization on multi-cpu and multi-gpu heterogeneous architectures. *The Journal of Supercomputing*, 70(2):756–771.

Pei, Y., Bosilca, G., and Dongarra, J. (2022). Sequential task flow runtime model improvements and limitations. In *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 1–8. IEEE.

StarPU Project (2015). *StarPU Handbook (version 1.4.8)*. Université de Bordeaux, CNRS, INRIA. Generated by Doxygen; GNU Free Documentation License.

Talib, M. A., Majzoub, S., Nasir, Q., and Jamal, D. (2021). A systematic literature review on hardware implementation of artificial intelligence algorithms. *The Journal of Supercomputing*, 77(2):1897–1938.

Tan, G., Shui, C., Wang, Y., Yu, X., and Yan, Y. (2021). Optimizing the linpack algorithm for large-scale pcie-based cpu-gpu heterogeneous systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2367–2380.

Tomov, S., Nath, R., Ltaief, H., and Dongarra, J. (2010). Dense linear algebra solvers for multicore with gpu accelerators. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE.

Valero-Lara, P., Kim, J., Hernandez, O., and Vetter, J. (2021). Openmp target task: Tasking and target offloading on heterogeneous systems. In *European Conference on Parallel Processing*, pages 445–455. Springer.

Xiao, W., Han, Z., Zhao, H., Peng, X., Zhang, Q., Yang, F., and Zhou, L. (2018). Scheduling cpu for gpu-based deep learning jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 503, New York, NY, USA. ACM.