EE-LSDS 2013

Energy Efficiency in Large Scale Distributed Systems conference

Runtime Scheduling of the LU factorization: Performance and Energy

Pedro Alonso[1], Manuel F. Dolz[2], Francisco D. Igual[3], Enrique S. Quintana-Ortí[2], and Rafael Mayo[2]

1  UNIVERSITAT POLITÈCNICA DE VALÈNCIA 2  UNIVERSITAT JAUME·I 3  UNIVERSIDAD COMPLUTENSE MADRID

April 22–24, 2013, Vienna, Austria

Introduction

- The power wall is a major hurdle that the scientific community will need to tackle in order to build the Exascale systems.
- In the last years, there exists a trend towards heterogeneous designs that combines hardware accelerators (GPUs) or DSPs.
- Programming a hybrid platform consisting of one to several multicore processors and multiple GPUs is still a considerable challenge.
- A number of runtime schedulers have been proposed in recent years to address the increase of hardware concurrency and, in some cases, the heterogeneity of recent architectures:
  - OmpSs and StarPU (task-level parallelism).
  - For DLA: SMPSs, StarPU, Quark, and `SuperMatrix`.

Introduction: SuperMatrix

SuperMatrix:

- Specifically designed from its inception for the execution of DLA operations.
- Allows the execution of the full functionality of the libflame DLA library on a variety of parallel platforms:
  - multicore desktop servers,
  - hybrid systems equipped with CPU-GPU, and
  - small-scale clusters.
- SuperMatrix runtime uses one CPU thread per GPU responsible of scheduling ready tasks for execution in the attached accelerator. This control thread runs in one CPU core, and no provision was made to exploit additional CPU cores in case they outnumbered the GPUs.

Introduction: Contributions

Contributions of this work:

- In the new runtime scheduler we accommodate one thread per CPU core in the system. Among these, there is one control thread per GPU, but now there is also one worker thread for each one of the remaining CPU cores. Therefore, in the new runtime scheduler we can leverage any combination of hardware CPU-GPU concurrency in the platform to increase performance.
- We introduce priorities in the runtime to advance the computation of tasks that lie on the critical path of the algorithm. As our experiments will illustrate, the outcome is a significant reduction of idle time, especially for hybrid CPU-GPU platforms, and faster execution.
- We integrate two energy-aware techniques into `SuperMatrix`.
- Finally, we analyze the practical impact of the performance enhancements and energy-saving strategies using the LU factorization with partial pivoting.

Outline

Algorithm: $[A] := \text{LU\_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

% **Factorize current panel**

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)\right] := \text{LU\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

% **Update trailing submatrix**

$A_{12} := \text{TRSM}(A_{11}, A_{12})$

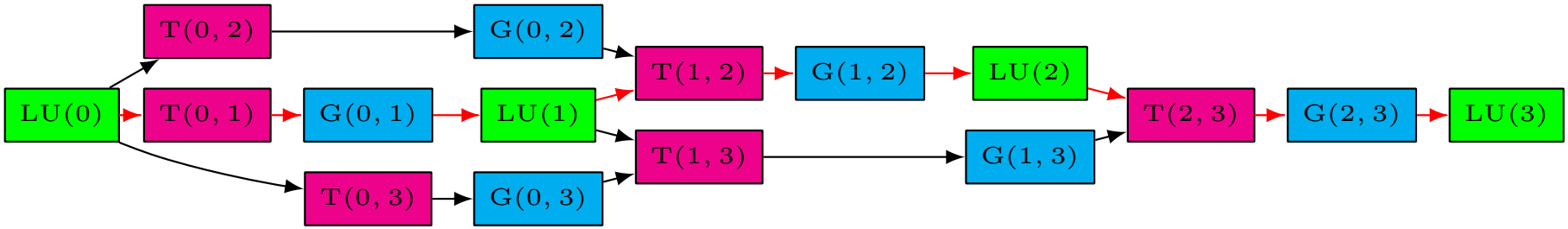
$A_{22} := \text{GEMM}(A_{21}, A_{12}, A_{22})$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Example of Task Dependency Graph



TDG for the LU factorization with partial pivoting of a matrix A consisting of $s \times s = 4 \times 4$ blocks. Red arrows identify the critical path of the algorithm.

The execution process

- Analysis stage:
  - The dependence analysis is performed by a single thread.
  - This thread inspects the code, inserting tasks as they are encountered into a data structure: work queue.
  - The root task is inserted directly into the ready queue.
  - The time for this stage is negligible.
- The working process:
  - A collection of worker threads poll the ready queue for new work.
  - A thread executes the corresponding computation.
  - This thread checks which dependencies have been fulfilled, moving those tasks with all dependencies satisfied to the ready queue.
  - The overhead introduced by the operation of the runtime is also negligible.

Particular details

The original version of **SuperMatrix** employs one control thread per GPU. These threads run on a (CPU) core of the host, and are in charge of

1. monitoring and updating the dependence queues;
2. guiding the associated accelerator by carrying out the necessary data transfers and dispatching tasks for execution there; and
3. executing computational work that is not suited to the GPU.

Software cache coherence:

- **SuperMatrix** considers GPU memories as full-associative caches and applies cache coherence policies (LRU, write-invalidate and write-update) between the main memory and the GPU memory.
- These techniques yield a significant reduction of the volume of communications between CPUs and GPUs
- This cache-memory coherence protocol in software has been proved to be efficient enough compared with with compute-intensive tasks which perform $O(b^3)$ arithmetic floating-point operations.

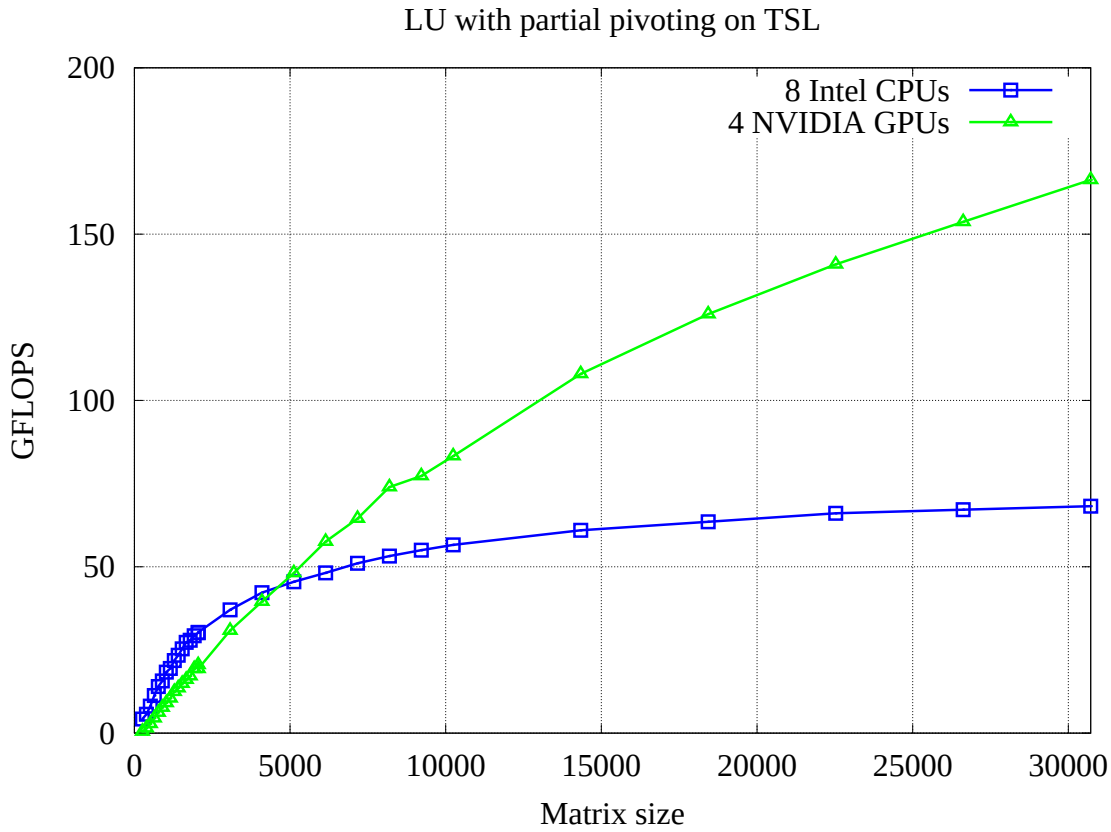
Environment Setup

- Hardware (TSL):
  - A server equipped with two Intel Xeon E5440 processors (4 cores per socket) at 2.83 GHz and 16 Gbytes of RAM, connected to
  - 4 "Fermi" GPUs NVIDIA Tesla S2050.
- Software
  - Intel MKL 10.0.1 for BLAS and LAPACK libraries, and
  - the `SuperMatrix` runtime in `libflame` release 5.0–r6719.
- The codes for the LU factorization were those from `libflame`.
- Energy measurement device: APC 8653 PDU (Power Distribution Unit) which samples power at 1 Hz.
- A daemon application ran on a separate tracing server, collecting power samples from the PDU (`pmlib`).

Performance analysis of the original SuperMatrix

The original **SuperMatrix** scheduler supported two different basic configuration modes:

- Multicore mode: One worker thread is bound to each CPU core, executing tasks on it by invoking optimized (sequential) BLAS/LAPACK kernels.
- MultiGPU mode: The scheduler performs the execution of tasks on platforms equipped with multiple hardware accelerators (GPUs).
  1. One control thread running on a CPU core is attached during the complete parallel execution to each GPU.
  2. Tasks are executed on the GPU using a specific implementation of BLAS for these devices, concretely, CUBLAS for NVIDIA GPUs.
  3. The CPU threads are in charge of performing the necessary data transfers between memory spaces prior to any task execution.
  4. In case a task is not appropriate for the GPU, the computation can be carried out in the associated CPU core.



Performance of the LU factorization with partial pivoting on TSL, using 8 CPU cores (multicore mode) and 4 GPUs (multiGPU mode)

Tuning the scheduler

In our extension of the **SuperMatrix** runtime for hybrid CPU-GPU architectures, decisions taken at runtime can address performance, but also energy efficiency:

1. The GFLOPS analysis in terms of the problem size determines an optimal mode for each problem size → the runtime system could modify, at execution time, the number of each type of computational resources (CPU or GPU) that are devoted to the actual task computations.
2. For medium-size matrices the attained performance is similar for two or more combinations (modes) of the number of CPUs/GPUs → the question thus becomes which mode is more efficient from the point of view of energy consumption.

Leveraging full hardware concurrency

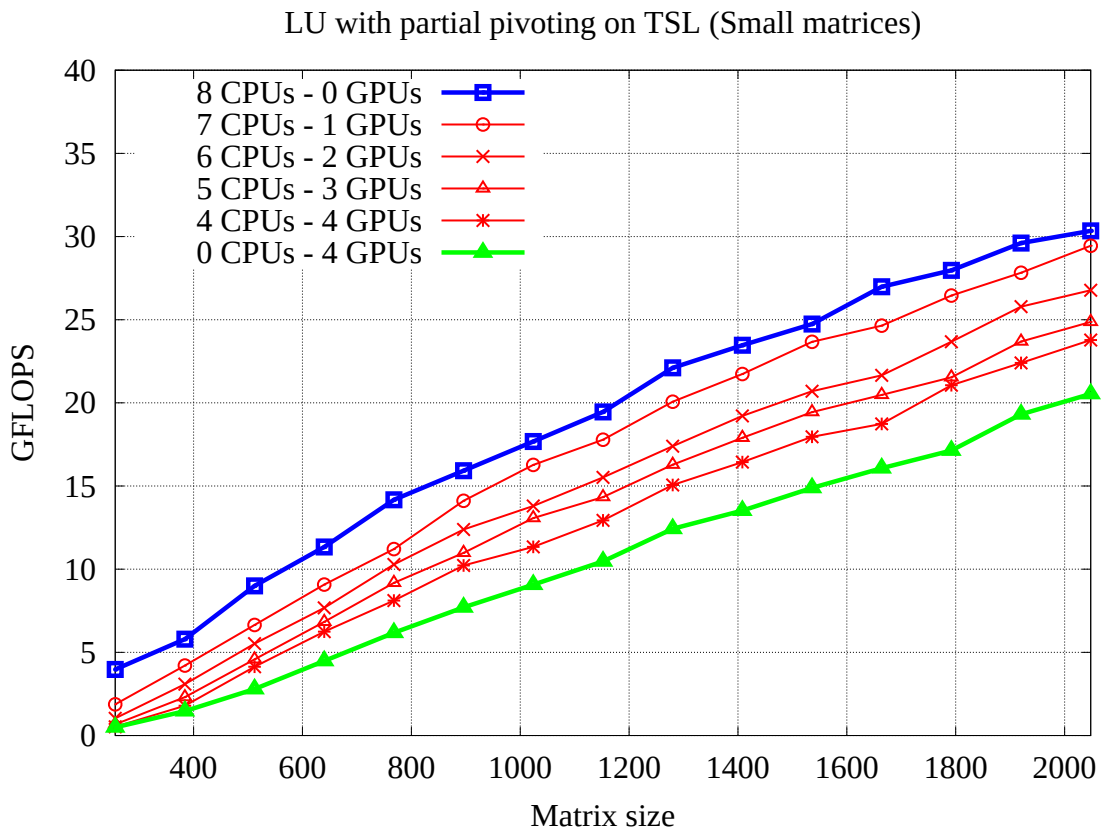
In the original SuperMatrix implementation, in the multiGPU mode

- a few types of tasks could be executed on the CPU cores → the corresponding GPUs remained stalled, waiting for the completion of the task.
- furthermore, only a number of cores = number of GPUs perform actual computational work.

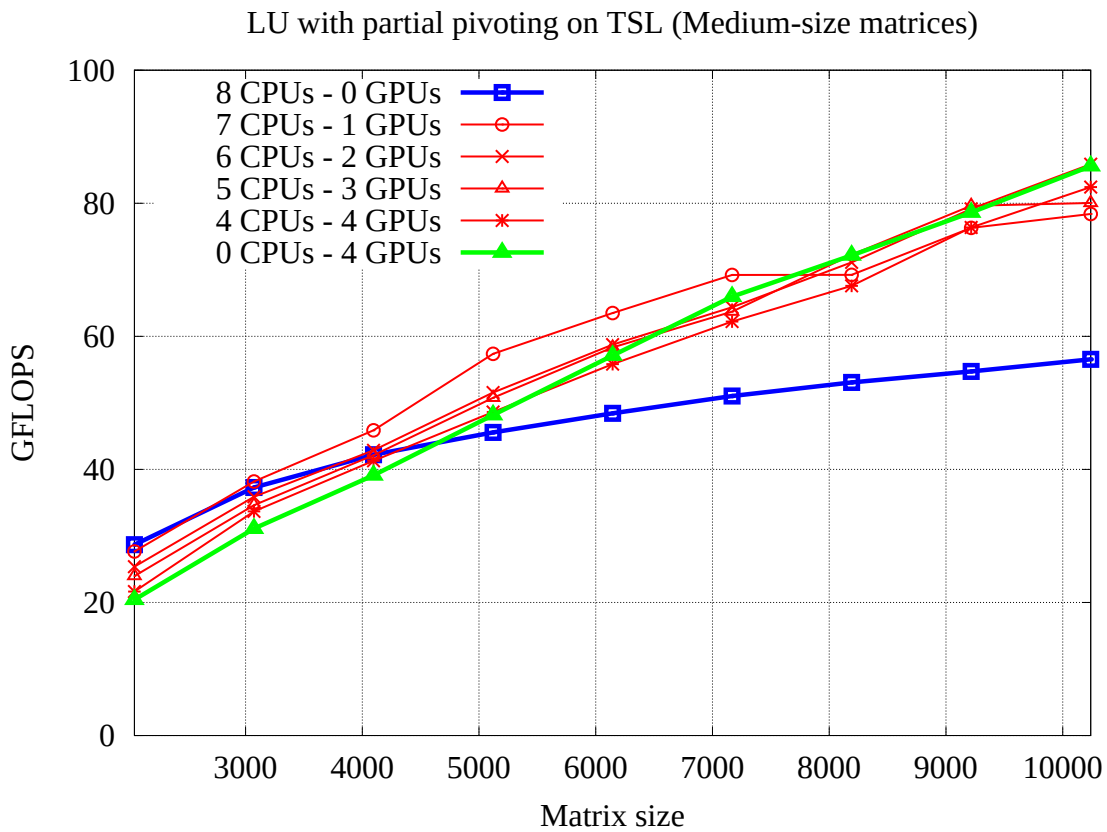
An improvement to this original execution model

- considers the GPUs and all the CPU cores as potential workers.
- In this case, each task type is bound to two different kernel instances, one for the GPU and one for the CPU.
- Depending on the type of thread a task is mapped to, the corresponding kernel instance is invoked.
- Data transfers are handled transparently by the runtime, depending on the type of worker thread and the location of the necessary data when the task is dispatched for execution.

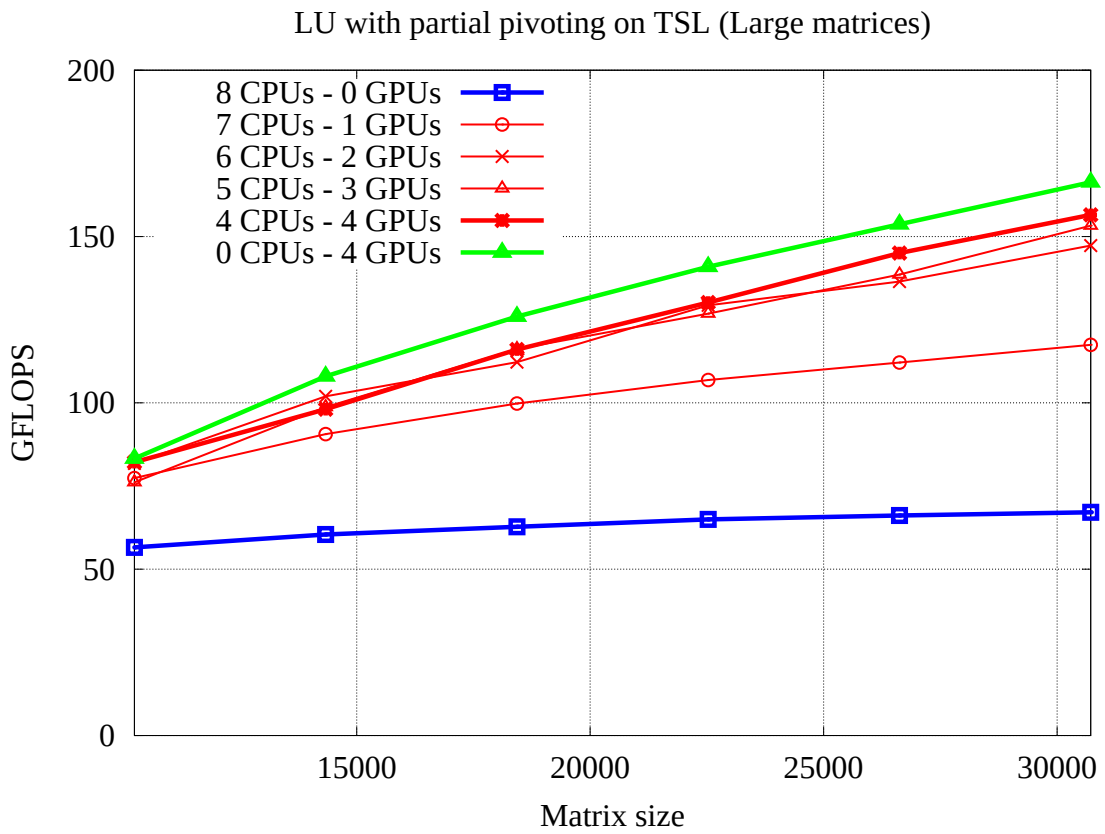
LU with small matrices



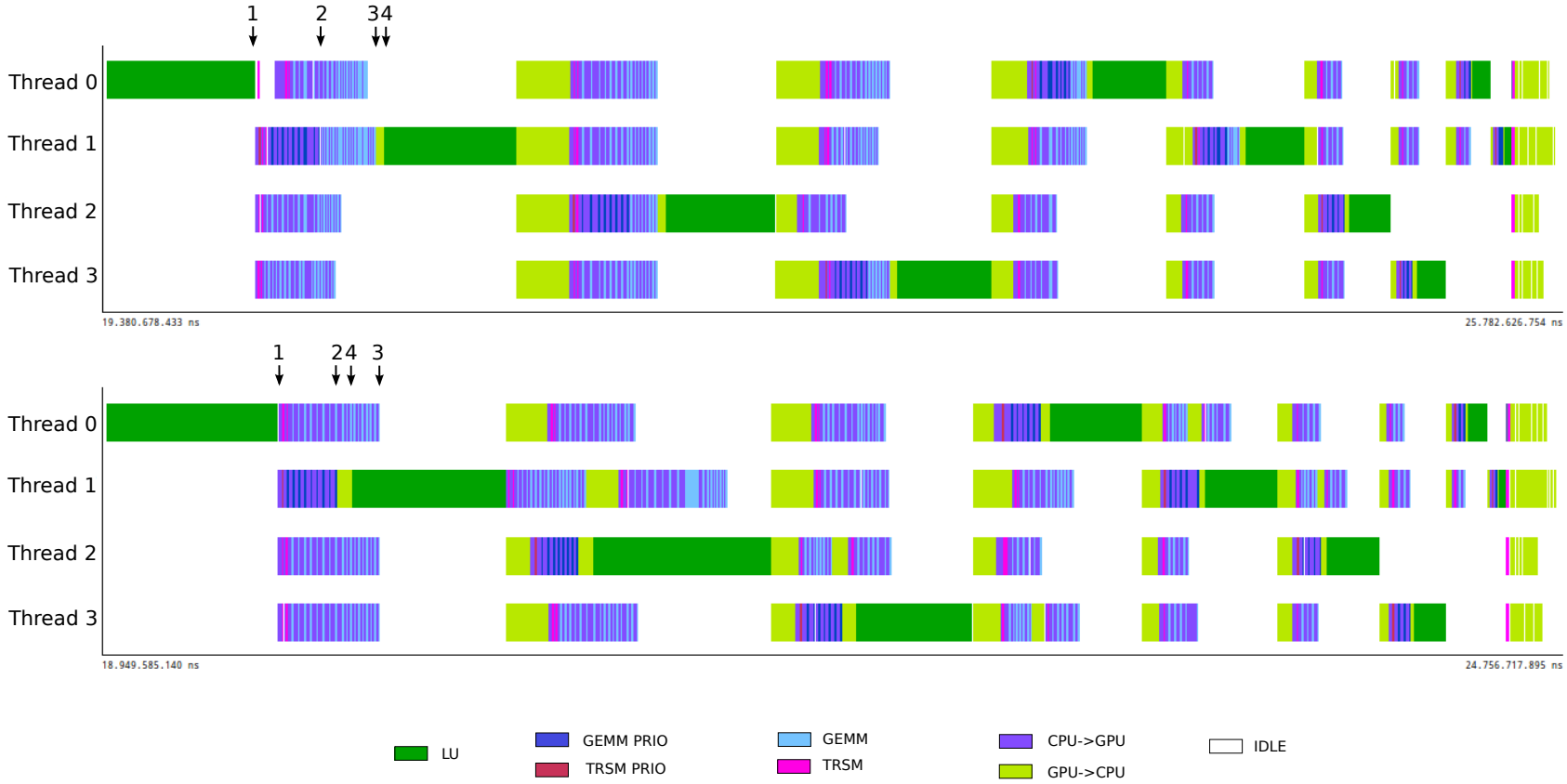
LU with medium matrices



LU with large matrices



Analysis with Extrae and Paraver



Traces of the execution of the LU factorization with partial pivoting of a matrix of dimension n=10,240, with b=1,024 using 4 GPUs of TSL, without priority tasks (top) and with priority tasks (bottom). Selected execution points: 1: End of LU(0); 2: End of UPDATE$(0, 1)$; 3: End of UPDATE$(0, s)$; 4: Start of LU(1).

Use of priorities

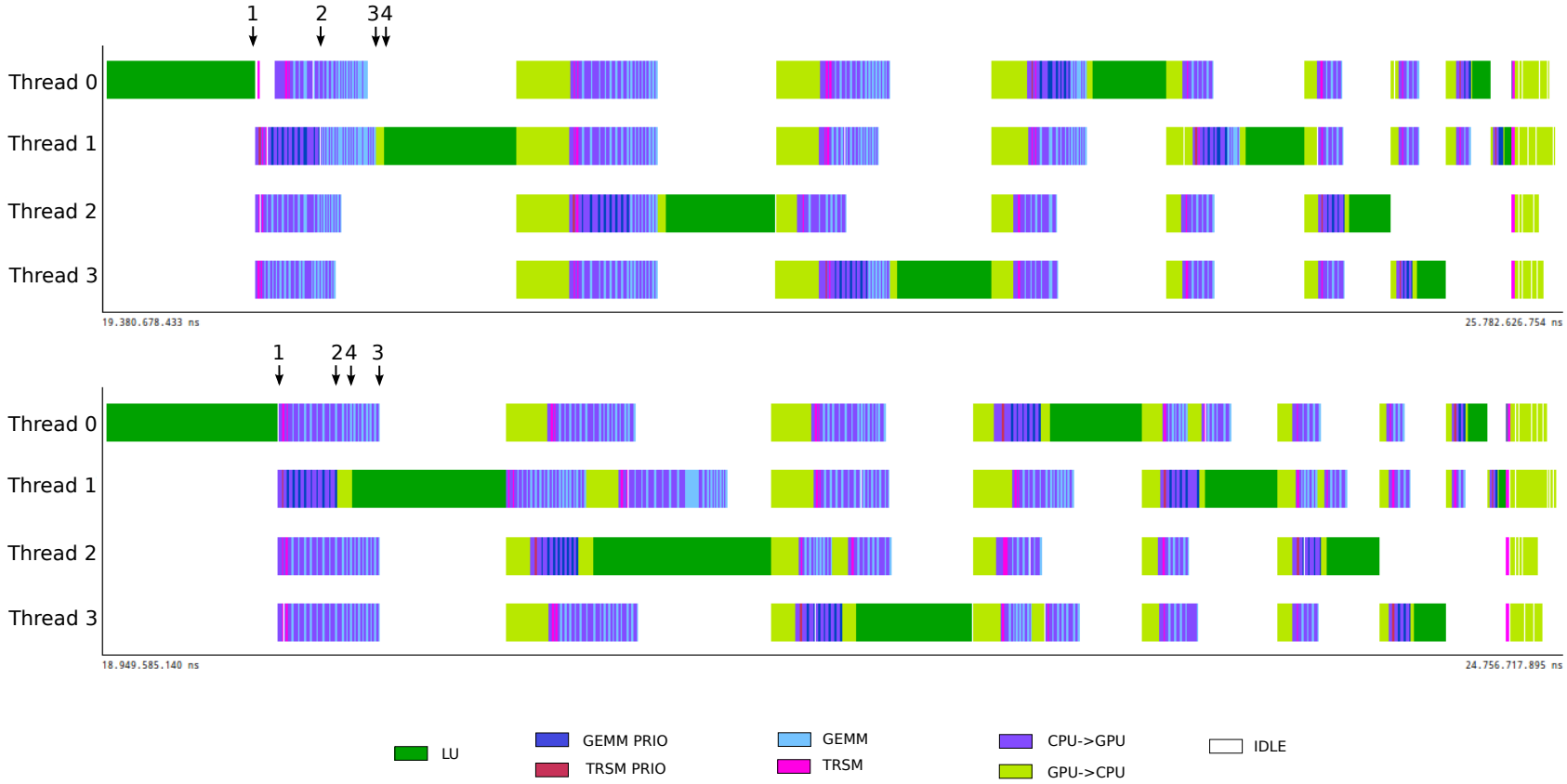
Tasks in the critical path receive a special treatment in the enhanced version of the **SuperMatrix** scheduler. Specifically,

1. critical (or priority) tasks are executed as soon as possible to avoid unnecessary stalls; and
2. they are mapped to the fastest computational resource (CPU or GPU) available.

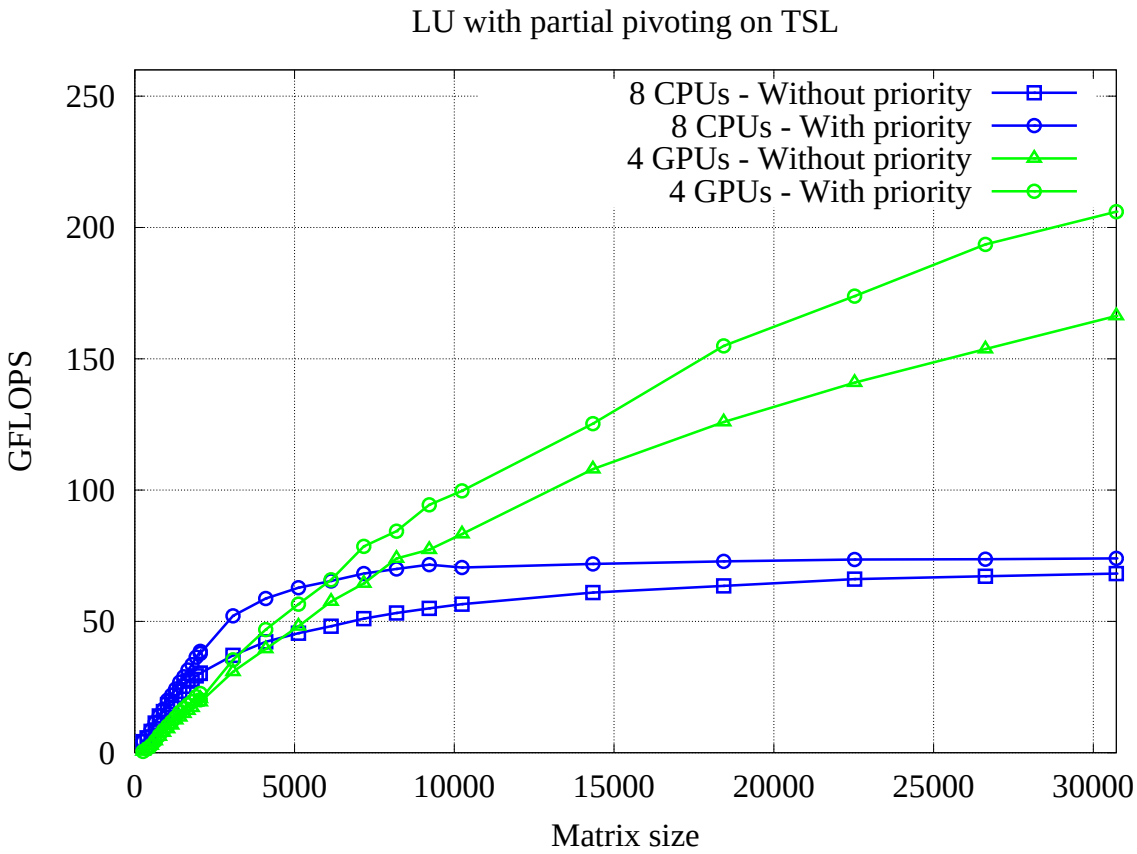
The scheduler has been modified to introduce an additional priority queue:

1. Ready tasks are removed from the work queue. If the task is critical, it is inserted in the thread's priority queue; otherwise, it is moved to a non-priority queue shared by all threads.
2. When a worker thread is idle, it polls the queues for ready tasks.
   - multiGPU mode: For threads binded to a GPU, they first check their priority queues; if no priority task is available, the shared non-priority queue is polled for a new task. If no GPU is attached, only the non-priority queue is polled.
   - multicore mode: The priority queue is always checked before the non-priority one.

Analysis with Extrae and Paraver



Traces of the execution of the LU factorization with partial pivoting of a matrix of dimension n=10,240, with b=1,024 using 4 GPUs of TSL, without priority tasks (top) and with priority tasks (bottom). Selected execution points: 1: End of LU(0); 2: End of UPDATE$(0, 1)$; 3: End of UPDATE$(0, s)$; 4: Start of LU(1).



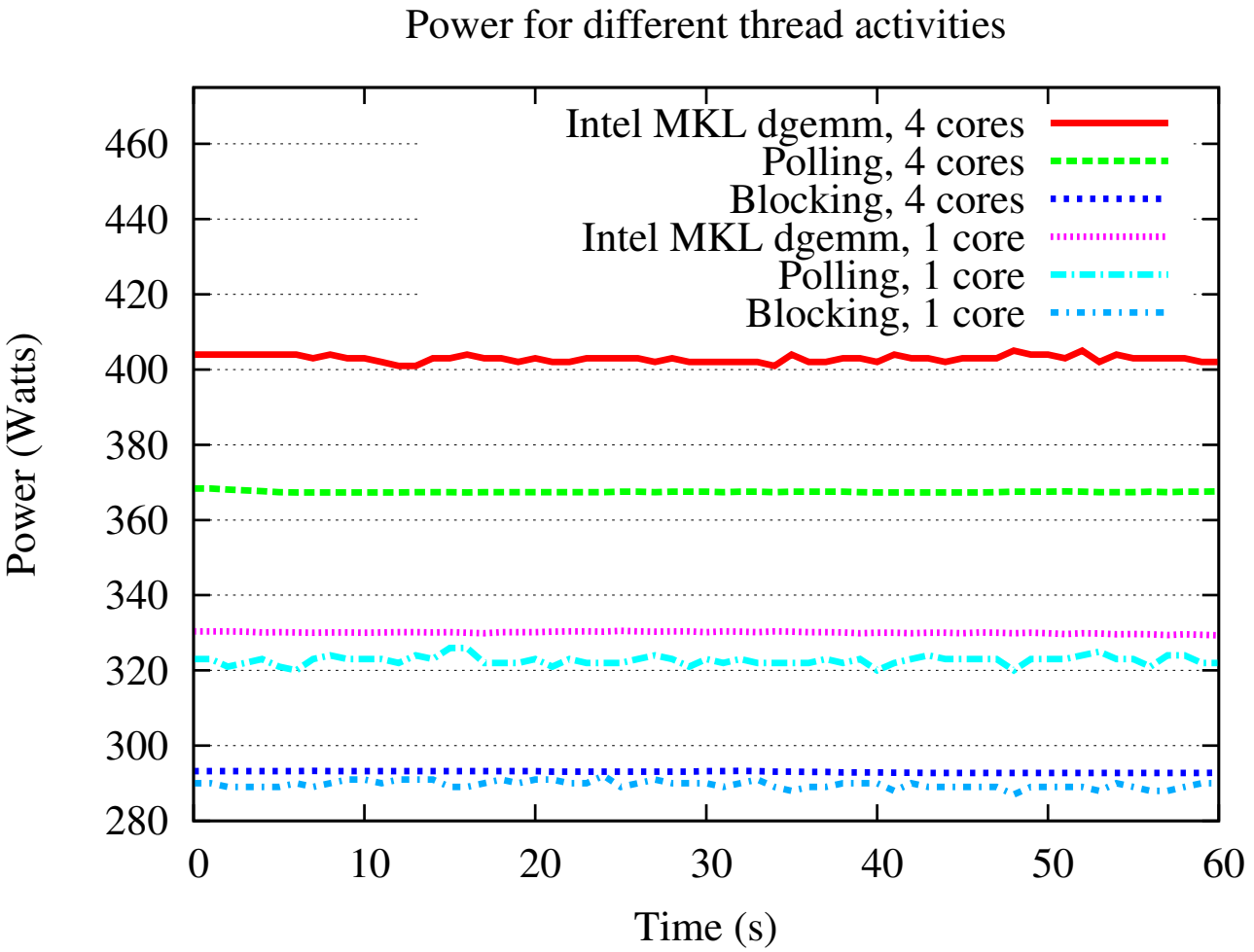
Impact of the use of priority tasks on the performance of the LU factorization on TSL, using 8 CPU cores (multicore mode) and 4 GPUs (multiGPU mode).

Following the former trace:

- Tasks of type LU are executed by the CPU cores, while the remaining two types of tasks are run on the GPUs.
- Idle time corresponds to periods when both the CPU cores and the GPUs perform no useful work (due to dependencies). These periods occupy a significant fraction of the execution.
- Additionally, note that when a GPU is working, the corresponding CPU core remains inoperative, waiting for the completion of the job.
- The question is how to leverage these inactive periods to reduce energy consumption.

Options to leverage idle time to save energy:

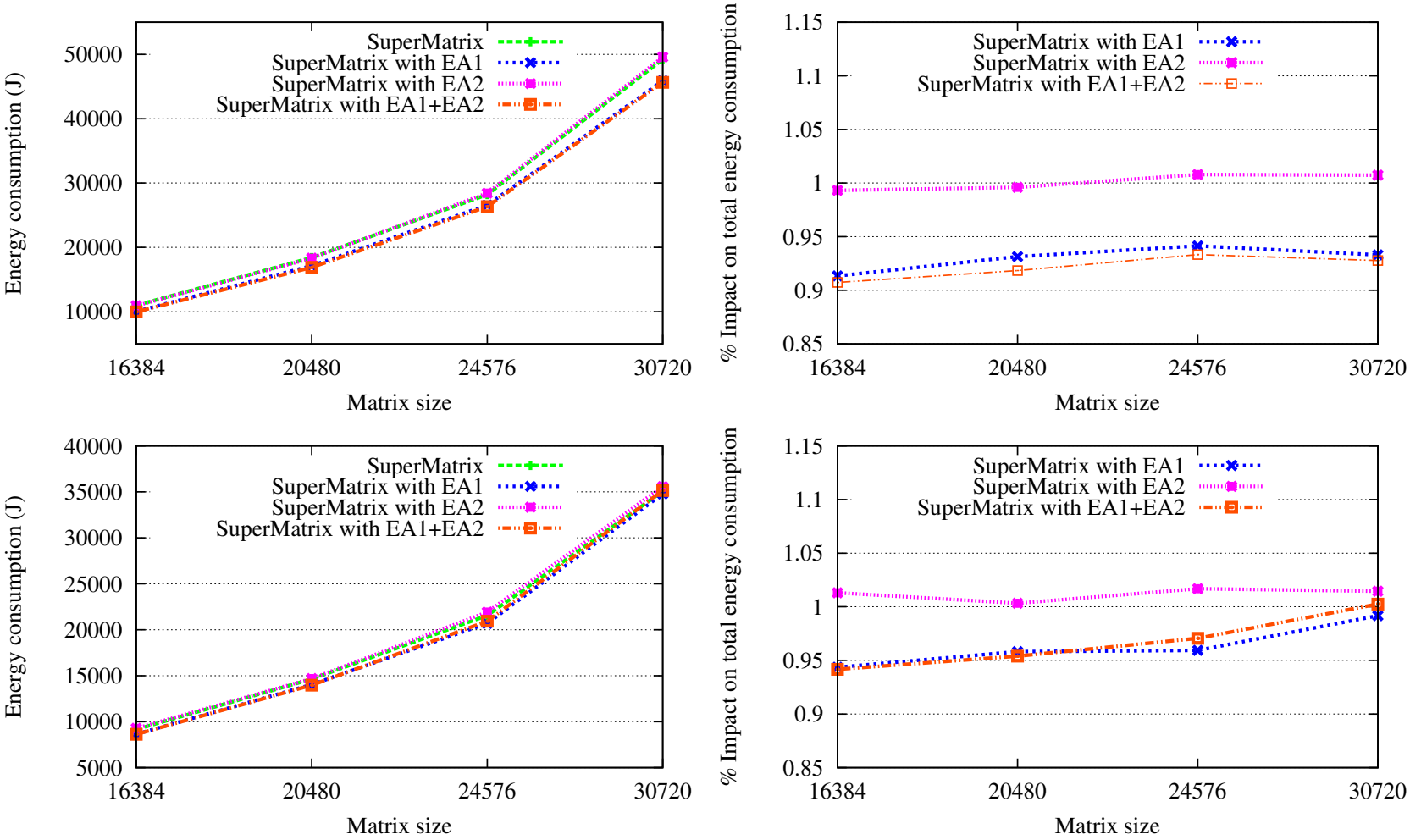
- DVFS (Dynamic Voltage and Frequency Scaling): our early experiments revealed that the gains attained by reducing the CPU frequency during the execution of compute-intensive DLA operations are small.
- Use "blocking" instead "polling".



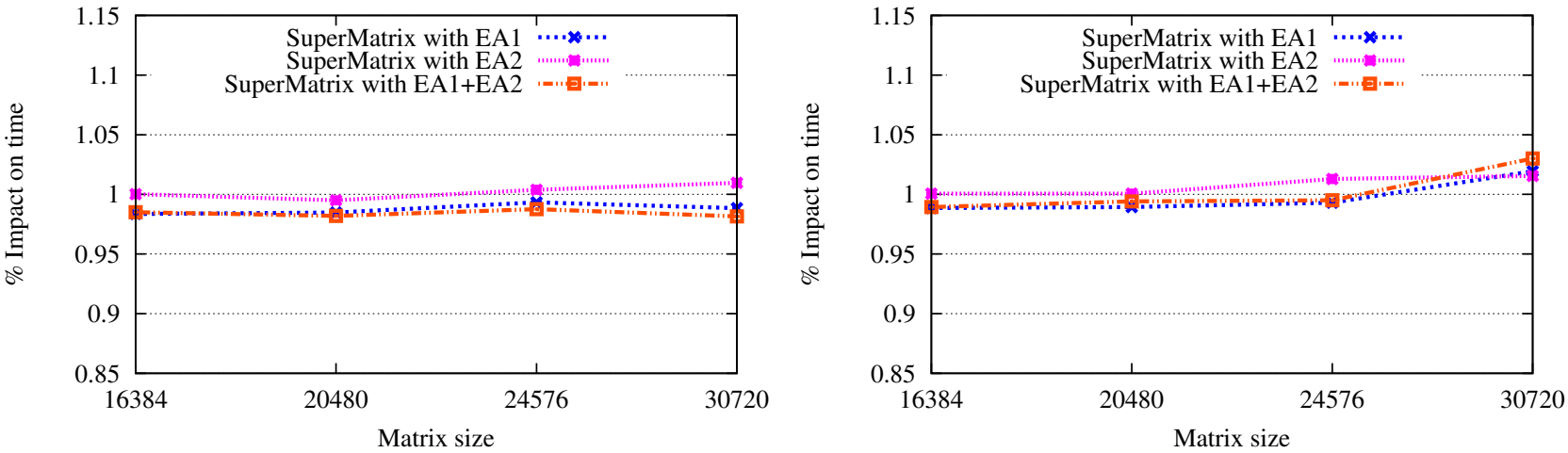
Power consumption of different actions performed by threads.

Techniques introduced in SuperMatrix

- Avoid polling when there are no ready tasks (EA1): we introduced POSIX semaphores into the runtime to control the activity of "idle" threads.
  - A CPU thread that polls the ready queue for a new task finds it empty blocks itself (`sem_wait()`).
  - An active CPU thread that completed its work and updated the dependencies, and in case this implies moving tasks from the work queue to the ready queue, this thread will wake up waiting threads (`sem post()`).
- Avoid polling when waiting for the GPU (EA2): A CPU core continuously dispatches tasks to its binded GPU. If the GPU is busy running a task, the thread enters a busy-wait until completion of the previous task.
  - By means of setting the parameter `cudaDeviceBlockingSync` invoking routine `cudaSetDeviceFlags` the CPU thread blocks the CPU thread on a synchronization primitive.
  - After activation, all synchronizations carried out using `cudaThreadSynchronize` will suspend the thread execution.



Impact on energy of the energy-aware techniques of the LU factorization with partial pivoting without and with priority tasks (top and bottom, respectively).



Impact on execution time of the energy-aware techniques of the LU factorization with partial pivoting without and with priority tasks (left and right, respectively).

Conclusions

In this paper we have introduced significant enhancements in the `SuperMatrix` runtime scheduler: the quest for high performance and the reduction of energy utilization.

- Performance:
  - Better exploitation of the hardware resources available in current CPU/GPU platforms.
  - Introduction and management of priority tasks to reduce in idle (20% of speedup).
- Energy: We have illustrated two different techniques that leverage idle times. This approach yields fair power savings for a minimal impact on the execution time.

These techniques are integrated into the runtime scheduler, and are transparent from the point of view of the developer.

Thanks for your attention!

Questions?