Improving Performance and Energy Consumption of Runtime Schedulers for Dense Linear Algebra

FLAME Working Note #73

Pedro Alonso*[1], Manuel F. Dolz†[2], Francisco D. Igual‡[3], Rafael Mayo§[4], and Enrique S. Quintana-Ortí¶[4]

[1]Depto. de Sistemas Informáticos y Computación, Universitat Politècnica de València, 46.022–Valencia, Spain
[2]Dept. of Informatics, University of Hamburg, 22.527–Hamburg, Germany
[3]Depto. de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, 28.040–Madrid, Spain
[4]Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, 12.071–Castellón, Spain

June 2, 2014

Abstract

The road towards Exascale Computing requires a holistic effort to address three different challenges simultaneously: high performance, energy efficiency, and programmability. The use of runtime task schedulers to orchestrate parallel executions with minimal developer intervention has been introduced in recent years to tackle the programmability issue while maintaining, or even improving, performance. In this paper, we enhance the SuperMatrix runtime task scheduler integrated in the `libflame` library in two different directions that address high performance and energy efficiency. First, we extend the runtime by accommodating hybrid parallel executions and managing task priorities for dense linear algebra operations, with remarkable performance improvements. Second, we introduce techniques to reduce energy consumption during idle times inherent to parallel executions, attaining important energy savings. In addition, we propose a power consumption model that can be leveraged by runtime task schedulers to make decisions based not only on performance, but also on energy considerations.

1 Introduction

With the introduction of the CUDA [1] and OpenCL [2] programming standards, graphics processing units (GPUs) are being increasingly adopted for their affordable price, favorable energy-performance balance and, due to their vast amount of hardware concurrency, the excellent acceleration factors demonstrated for many compute-intensive applications with ample data-parallelism [3, 4]. Nevertheless, this type of hardware accelerators has to be attached to a conventional (multicore) processor (or CPU), and efficiently programming a heterogeneous platform consisting of one to several multicore processors and multiple GPUs is still a considerable challenge. The reason is that, when dealing with these parallel (hybrid) systems, in addition to facing the programming difficulties intrinsic to concurrency, the developer has to cope with the existence of multiple memory address spaces, and the different programming models.

*palonso@dsic.upv.es
†dolzm@icc.uji.es
‡figual@fdi.ucm.es
§mayo@icc.uji.es
¶quintana@icc.uji.es

In recent years, a number of *runtimes* have been proposed to alleviate the burden that programming these new platforms with increased levels of thread parallelism pose. Thus, OmpSs [5], StarPU [6], Mentat [7] and Harmony [8], among others, have followed the approach pioneered by Cilk [9], offering implicit parallel programming models with dependence analysis, especially well adapted to exploit task-level parallelism while partially palliating the programmability problem. When applied to Dense Linear Algebra (DLA) operations in particular, SMPSs (a precursor of OmpSs) [10], StarPU, Quark [11] and SuperMatrix [12] have demonstrated the advantage of extracting task-level parallelism using this same approach for this specific domain.

On the other hand, heterogeneous architectures that combine general-purpose multicore technology with hardware accelerators like GPUs or the Intel Xeon Phi, DSPs (digital signal processors), and FPGAs (field programmable gate arrays), also seem to provide the answer to the continued pressure to further reduce power consumption [13, 14] historically exerted by the mobile and embedded appliances, but now also in place for HPC facilities and datacenters [15].

The SuperMatrix runtime was designed from its inception for the execution of DLA operations. This runtime follows to the methodology advocated in the FLAME project [12], which patronizes a separation of concerns between the derivation of new algorithms for DLA operations and their practical coding (implementation) and high-performance execution on a platform. SuperMatrix orchestrates a seamless, task-parallel execution of the full functionality of the `libflame` DLA library [16] on a range of platforms, including multicore desktop servers [17], heterogeneous CPU-GPU systems [18], and small-scale clusters [19].

In this paper, we present a few major changes to SuperMatrix, that yield significant improvements on both performance and energy consumption. In particular, we make the following contributions:

- The original SuperMatrix runtime commits one CPU thread per GPU that schedules tasks for their execution in the accelerator attached to it. This control thread monopolizes one CPU core, and no attempt is made to exploit additional CPU cores in case these outnumber the GPUs. In the new version of the runtime we instead accommodate one thread per CPU core in the system. Among these, there is one control thread per GPU, but now also one additional worker thread for each one of the remaining CPU cores, which allows the new runtime to leverage any combination of hardware CPU-GPU concurrency in the platform.

- During the experimental evaluation of certain dense matrix decompositions on heterogeneous CPU-GPU platforms, it was observed that the panel factorization that lies in the critical path of the algorithm is a major obstacle to attain high performance. To deal with this problem, we introduce task priorities in the new runtime, to advance the computation of these operations. The experimental results show a significant reduction of idle time in this type of platforms.

- Furthermore, we integrate two energy-aware techniques into the new SuperMatrix runtime and we formulate a model of the energy consumption for DLA operations that allows us to relate the experimental energy savings with the theoretical expectations.

- Finally, we analyze the actual impact of the performance and energy-saving enhancements using two key DLA operations, namely the LU factorization with partial pivoting and the Cholesky decomposition, that are representative of many other dense Level-3 BLAS-based matrix operations.

While some of these results were already presented in [20], the detailed description of how to introduce priorities into the runtime, the experimental evaluation of the Cholesky factorization, both from the perspective of performance and energy consumption, and the power consumption model are new contributions specific to this paper.

The rest of the paper is structured as follows. In Section 2 we briefly review the foundations of data-flow runtimes and the operation of SuperMatrix. In Section 3 we describe the target platform, the computational libraries that are employed, and the power measurement setup. Sections 4 and 5 contain the major contributions of our paper, the performance enhancements and the energy-aware mechanisms incorporated into the runtime, respectively. A few remarks and a discussion of future work close the paper in Section 6.

2 SuperMatrix Data-Flow Parallel Runtime for Dense Linear Algebra

In this section, we describe the internals of the SuperMatrix runtime task scheduler by using two well-known basic DLA operations: the LU with partial pivoting and the Cholesky factorizations. These two operations will drive the explanation throughout the rest of the document. Although we focus on the LU factorization to guide the description of our runtime, similar ideas underlie the Cholesky factorization in particular, and other DLA operations in general.

2.1 A brief introduction to SuperMatrix's data-flow execution

Consider the LU factorization of a (nonsingular) matrix $A \in \mathbb{R}^{n \times n}$, which computes the decomposition $A = LU$, where $L \in \mathbb{R}^{n \times n}$ is unit lower triangular and $U \in \mathbb{R}^{n \times n}$ is upper triangular. For simplicity, we do not consider pivoting during the presentation, though all the implementations evaluated in this paper include this technique. Figure 1 (left) presents a right-looking blocked algorithm for this factorization using the FLAME notation [21].

Supermatrix (like many other high performance runtimes for DLA) starts from a (sequential) blocked algorithm of the target matrix operation (see Figure 1), to obtain a task parallel data-flow execution. For this purpose, SuperMatrix first decomposes the algorithm/operation into a number of suboperations (*tasks*) of a certain granularity, while simultaneously identifying all dependencies among these. In the case of SuperMatrix and DLA operations, this can be done, e.g., based only on the order in which tasks appear in the algorithm as well as the operands that each task reads (inputs), writes (outputs), or reads/writes (inputs/outputs).

For example, consider an $n \times n$ matrix A composed of $s \times s = 4 \times 4$ blocks of dimension $b \times b$ each (i.e., $n = s \cdot b$). The symbolic result from the above process is the task dependency graph (TDG) in Figure 2, where $\mathsf{LU}(k)$ stands for the factorization of the k-th panel (column block), and $\mathsf{T}(k, j)$ and $\mathsf{G}(k, j)$ refer, respectively, to the triangular system solve and the matrix-matrix update of the j-th panel with respect to the factorization of panel k (see Figure 1 (right)). In this operation, the factorization of the first panel, $\mathsf{LU}(0)$, yields a result that is necessary for tasks $\mathsf{T}(0, 1)$, $\mathsf{T}(0, 2)$, $\mathsf{T}(0, 3)$; and this is captured in the TDG by arcs (dependencies) between the corresponding nodes (tasks). Thus, these dependencies state that $\mathsf{T}(0, 1)$–$\mathsf{T}(0, 3)$ cannot be executed till $\mathsf{LU}(0)$ is completed, but also that these three triangular solves can be performed in any order.

Following a similar approach, we consider a blocked algorithm for the Cholesky factorization, shown in Figure 3, that decomposes a symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ into the product $A = LL^T$, where $L \in \mathbb{R}^{n \times n}$ is lower triangular. The symbolic analysis of the algorithm for a matrix composed of $s \times s = 4 \times 4$ blocks, of dimension $b \times b$ each, yields the TDG in Figure 4, where CHOL, T, S, and G stand for the Cholesky factorization, triangular system solve, symmetric rank-b update, and matrix-matrix product, respectively.

In summary, the TDG associated with a given algorithm dictates different "orderings" in which the tasks (suboperations) can be correctly computed, and SuperMatrix leverages this information to produce an out-of-order, data-flow schedule of the TDG and a concurrent execution of the tasks.

2.2 Operation of SuperMatrix

After the identification of tasks and dependencies, SuperMatrix proceeds to execute the computations represented by the TDG. For that purpose, the runtime spawns a collection of worker threads that poll a queue of tasks *ready* for execution. Upon dequeuing a task from this structure, a thread executes the corresponding computation and, once completed, checks which dependencies have been fulfilled, moving those tasks with all dependencies satisfied from the global work queue to the ready queue.

The original version of SuperMatrix for heterogeneous CPU-GPU platforms [18] commits one control thread per GPU (device) of the target platform. These threads run each on a different (CPU) core of the host, and *i)* update the dependence queues; *ii)* guide the associated accelerator by carrying out the necessary data transfers and dispatching tasks for execution there; and *iii)* execute computational work that is not suited to the GPU. For example, for the LU factorization with partial pivoting, the panel factorizations are performed by the CPU cores, as this type of operations requires a fine control that renders them inappropriate

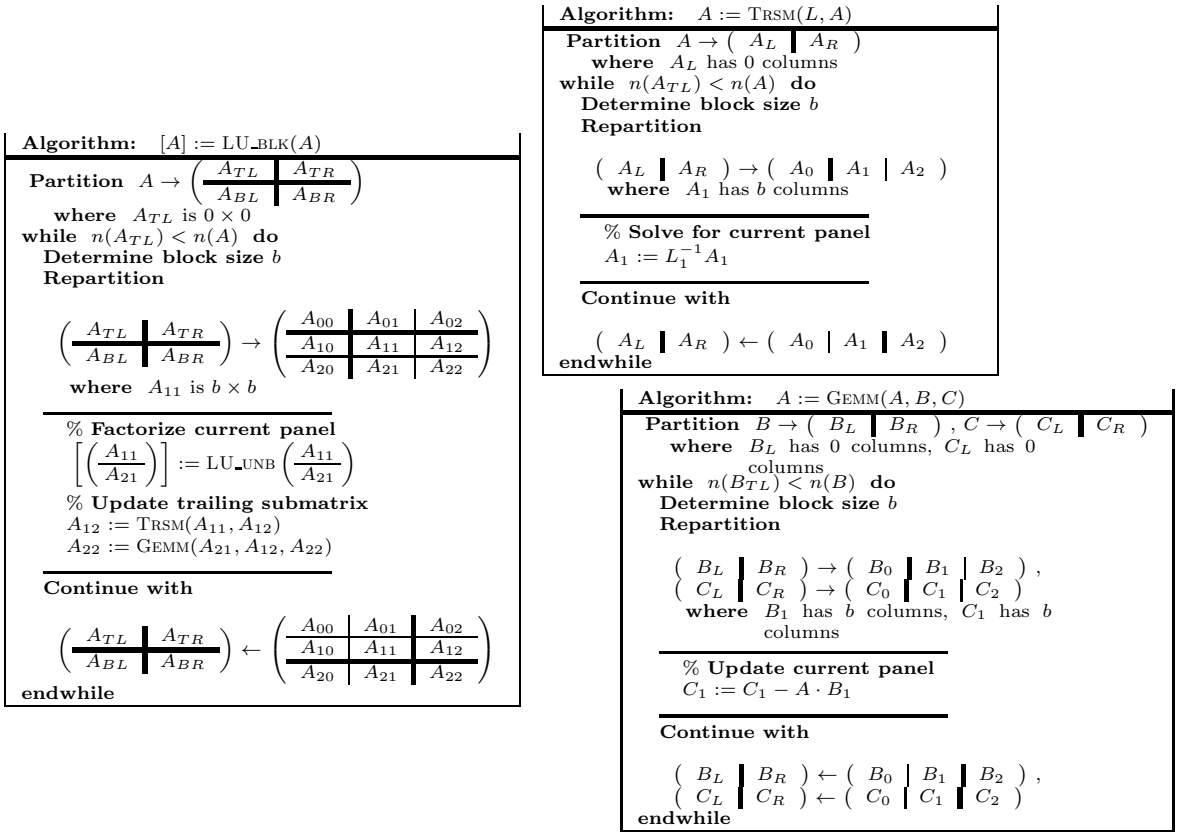


Figure 1: Blocked algorithm for the LU factorization (left) and procedures for the update of the trailing submatrix (top and bottom right). In the algorithms, procedure $\text{LU\_UNB}(\cdot)$ computes the LU factorization of a panel using an unblocked algorithm, and the operator $n(\cdot)$ returns the number of columns of its argument.

for the GPU. The triangular solves (T) and matrix-matrix updates (G) of the remaining blocks, on the other hand, are performed by the GPUs. A major drawback of that initial version is that no attempt was made to exploit the existence of more CPU cores than GPUs in the target platform.

For the particular case of parallel platforms with multiple memory address spaces, this version of the runtime [18] introduces two communication-reducing techniques: *i)* the workload is partitioned statically among the computational resources following a cyclic block data layout; and *ii)* the memory of each GPU is viewed as a local, fully-associative cache, and data coherence is preserved using *write-invalidate* and *write-back* protocols [22]. The outcome is a significant reduction of the volume of communications between CPU and GPU, diminishing the impact of the slow PCI-e bus.

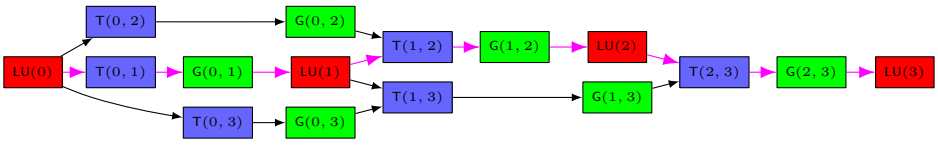


Figure 2: TDG for the LU factorization with partial pivoting of a matrix A consisting of $s \times s = 4 \times 4$ blocks. Pink arrows identify the critical path of the algorithm.

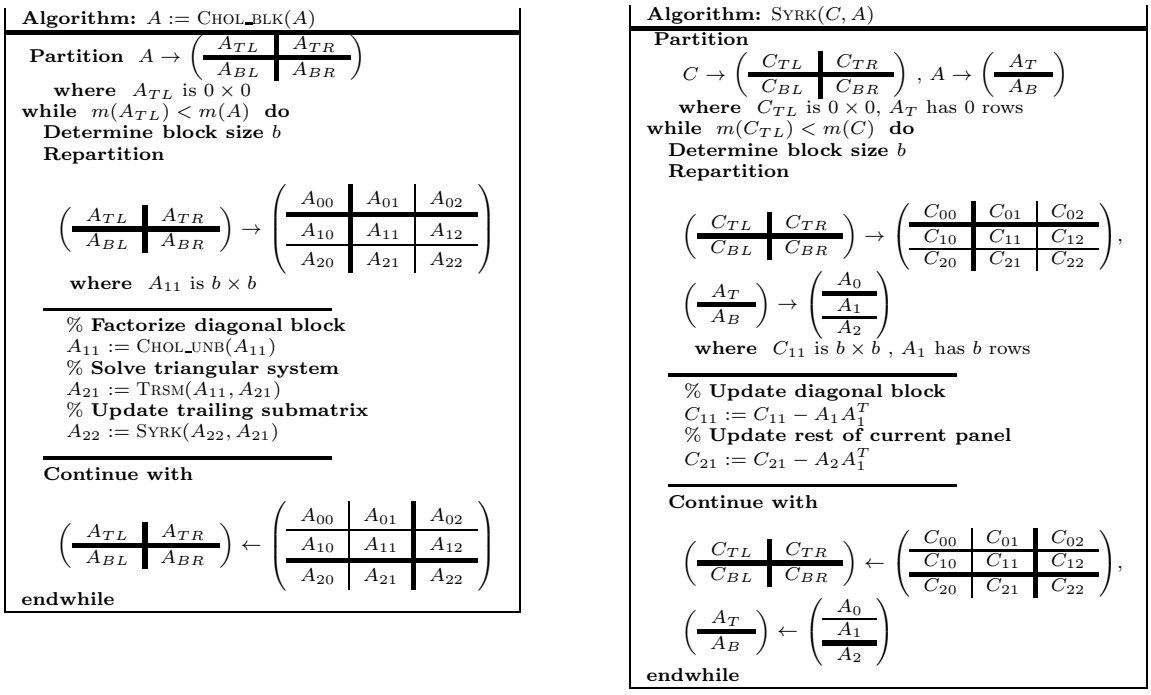


Figure 3: Blocked algorithm for the Cholesky factorization (left) and procedure for the update of the trailing submatrix (right). Procedure $\text{CHOL\_UNB}(\cdot)$ computes the Cholesky factorization of a block using an unblocked algorithm, and operator $m(\cdot)$ returns the number of rows of its argument. Blocked algorithms for the solution of a triangular system and matrix-matrix product are similar to those shown in Figure 1.

3 Environment Setup

In order to compare the different versions of the runtime we used two servers. The first one (called TSL1 hereafter) is equipped with two Intel Xeon E5440 processors at 2.83 GHz featuring 4 cores per socket each, 16 Gbytes of RAM and is connected to 4 "Fermi" GPUs NVIDIA Tesla S2050. The second server (TSL2) is composed of two quadcore Intel Xeon 5405 processors at 2.00 GHz, with 24 Gbytes of RAM and is connected to a NVIDIA Tesla S1070 (4 Nvidia Tesla C1060 GPUs). Both servers use the same software: Intel MKL 10.0.1 on the CPU side and NVIDIA CUBLAS 5.0 on the GPU side for BLAS/LAPACK operations (IEEE double-precision real arithmetic), and the SuperMatrix runtime in `libflame` (release 5.0–r6719) for the factorization routines. In the execution of the routines we use the optimal value for the block size, which was selected via a complete experimental analysis.

The power supply units of the servers are connected to an APC 8653 Power Distribution Unit (PDU). A separate tracing server runs a daemon application that samples power from the PDU at rate 1 Hz, using the routines from our `pmlib` [23] library to interact with the power measurement device. These routines allow to synchronize the target application and the daemon on the tracing server, sample the wattmeter, and dump the power figures into a file upon termination of the application.

4 Improving SuperMatrix

In this section, we first propose an initial experiment to gain insights about the performance of the SuperMatrix scheduler using the two only configurations available in its original design: the multicore and multiGPU setups. The results in Section 4.1 justify the introduction of improvements to the scheduler in order to leverage the full potential of the hardware resources available in current hybrid CPU-GPU architectures, by further exploiting the existing concurrency in DLA operations. Techniques and necessary modifications

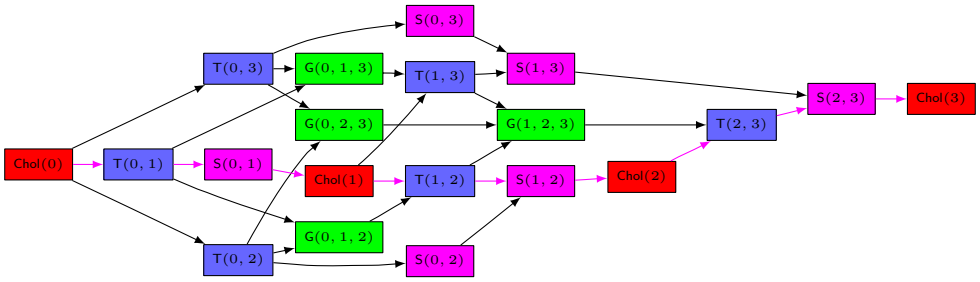


Figure 4: TDG for the Cholesky factorization of a matrix A consisting of $s \times s = 4 \times 4$ blocks. Pink arrows identify the critical path of the algorithm.

applied to the scheduler are described in Sections 4.2 to 4.4.

4.1 Performance analysis of the original SuperMatrix

The original design of the SuperMatrix scheduler for heterogeneous CPU-GPU architectures supported two basic execution modes, depending on the available hardware resources:

- *Multicore mode.* In this setup, one worker thread is bound to a unique CPU core, executing tasks on it via invocations to optimized (sequential) BLAS/LAPACK kernels, and collaborating in the management of shared lists of ready and pending tasks.

- *MultiGPU mode.* In this configuration, the runtime task scheduler executes tasks on platforms equipped with one or more hardware accelerators attached —specifically GPUs— transparently to the user/library developer. One control thread running on a CPU core is associated to a different GPU during the complete parallel execution. As the computation advances, ready tasks are executed in the GPUs using a specific implementation of BLAS for these devices (in our case, CUBLAS for NVIDIA GPUs). The control threads are in charge of performing the necessary data transfers between memory spaces prior to any task execution in the GPU, if necessary. In case a task is not appropriate for the GPU, the computation can be carried out in the associated CPU core, and no data transfers are performed unless transfers are strictly required to maintain data consistency.

Let us next illustrate the performance of the original modes of the SuperMatrix scheduler using our driving examples of the LU factorization with partial pivoting and the Cholesky factorization. Figure 5 shows the performance of the codes for these operations, in terms of GFLOPS (i.e., billions of floating-poing arithmetic operations, or flops, per second), using 8 CPU cores (multicore mode) and 4 GPUs (multiGPU mode) of TSL1.

From this initial experiment, it is possible to extract a few coarse conclusions, identifying three different cases according to the matrix dimension:

- *Small problems.* For small matrices (up to $n = 4,500$ for the LU factorization and up to $n = 5,500$ for the Cholesky factorization), the multicore mode outperforms the performance of the GPU-based alternative. This is a known result [24, 18] as GPUs need large volumes of computation in order to hide PCI-e data transfer overheads and to exploit the massive hardware concurrency featured by the accelerator.

- *Medium-size problems.* For medium-size matrices (in the range $n = 4,500$ to $5,500$ for the LU factorization and $n = 5,500$ to $6,500$ for the Cholesky factorization), both the multicore and multiGPU modes deliver similar GFLOPS rates. This behavior, in which equivalent performance for different configuration modes appears, is common to other DLA operations, and justifies the derivation of a power model to determine the optimal mode from the point of view of energy consumption.

- *Large problems.* For large matrices (starting at $n = 5,500$ for the LU factorization and $n = 6,500$ for the Cholesky factorization), the multiGPU setup clearly outperforms its multicore counterpart,

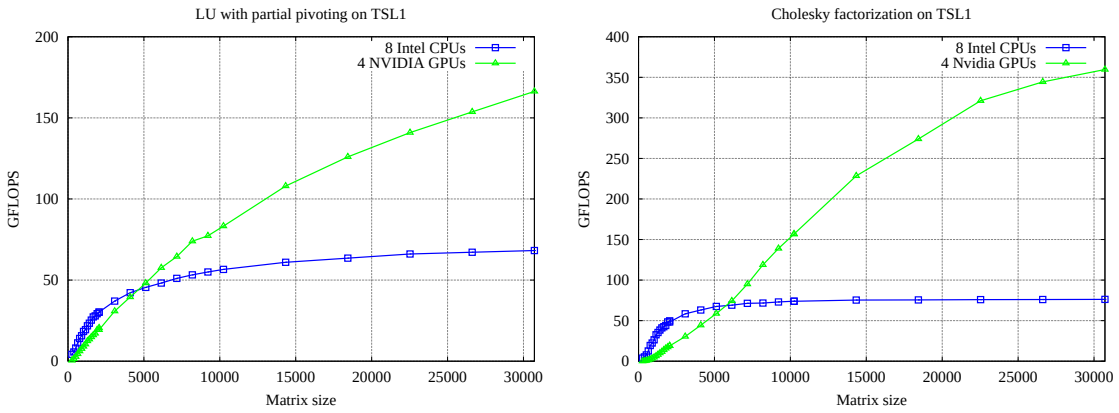


Figure 5: Performance of the LU factorization with partial pivoting (left-side plot) and Cholesky factorization (right-side plot) on TSL1, using 8 CPU cores (multicore mode) and 4 GPUs (multiGPU mode).

basically due to the much higher hardware concurrency of the GPUs. The gap in performance between both platforms becomes larger as the problem size increases.

4.2 Tuning the scheduler

In past work, a number of heuristics have been applied to tune the performance of runtime task schedulers by reducing idle time and/or minimizing data movement among memory spaces. These techniques include data affinity [18], caching [25], work-stealing [26], or task renaming [27].

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

- The GFLOPS analysis in terms of the problem size reported in the previous subsection determines an optimal mode for each problem size. To leverage this, 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 computation. This type of decisions is even more important and complex for a hybrid runtime implementation (alike that presented next), in which any combination of number of CPU/GPU worker threads can be chosen at runtime.

- For the range of matrices labeled as *medium-size problems* in the previous subsection, the attained performance is similar for two or more combinations of the number of CPUs/GPUs. The question thus becomes which mode is more efficient from the point of view of energy consumption. To answer this, in Section 5 we propose a power model for hybrid CPU/GPU architectures, which could be leveraged by runtime schedulers to make decisions at execution time on the number of resources that should be utilized in order to minimize energy consumption in situations in which performance is expected to be similar, but power consumption may dramatically vary depending on the number of CPU/GPU worker threads.

The next two subsections address the performance goal while the discussion related to energy is delayed till Section 5.

4.3 Leveraging full hardware concurrency

In the original SuperMatrix implementation, the execution of tasks was performed either by the CPU cores (multicore mode) or by the GPUs (multiGPU mode). As an exception, in the multiGPU mode a few types of tasks could be executed on the CPU cores, due to their special properties. While this occurred, though, the corresponding GPUs remained stalled, waiting for the completion of the task. As a result, for example, if four GPUs were used from a platform equipped with 12 CPU cores in the multiGPU mode, only the four

cores assigned to guide the execution on the GPUs were effectively utilized, while the remaining eight CPU cores were wasted.

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 by the runtime, depending on the type of worker thread and the location of the necessary data when the task is dispatched for execution.

Figure 6 reports the performance of the LU factorization with partial pivoting and the Cholesky factorization using the modified scheduler that accommodates hybrid executions. In the plots, red lines identify hybrid configurations, while blue and green lines correspond to multicore and multiGPU configurations, respectively. We illustrate the results by dividing the overall performance lines into three different cases, depending on the problem size; each case is shown in a different pair of plots.

- *Small problems (Figure 6(a)).* The addition of GPUs to the basic multicore setup does not improve performance for these problem sizes. Hybrid configurations perform better than the multiGPU one, offering increasing levels of performance as the number of GPUs decreases. From these results, it is clear that the progressive activation of GPUs for small problems degrades performance, and only the CPU cores should be used for these particular problem sizes.

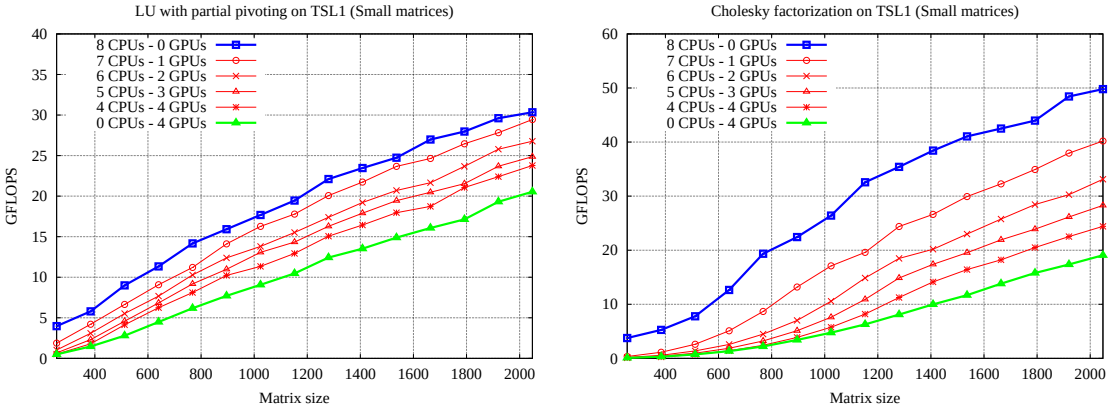
- *Medium-size problems (Figure 6(b)).* The insights for these particular problem sizes involve both performance and energy-related considerations. Regarding *performance*, hybrid configurations perform better than multicore and multiGPU configurations for most problems sizes in the range. This fact offers the hybrid scheduler different options to select the most appropriate configuration mode at runtime to tune performance for a given problem size. Regarding *energy consumption*, note that situations in which performance lines for different configurations intersect each other appear frequently in the plot. For these problem sizes, performance for different configurations is approximately equivalent. For example, consider the LU factorization and compare the performance attained by the multicore configuration and a hybrid setup using 7 CPU cores/1 GPU for n = 3,072. While the performance attained is barely identical, the efficiency of the execution for both configurations is likely to vary significantly when employing or not the GPU. In this type of situations, the hybrid scheduler can make decisions based on energy-aware considerations, selecting the most suitable execution configuration at runtime.

- *Large problems (Figure 6(c)).* In this case, the situation is the opposite to that for small problems: the addition of GPUs to the basic multicore setup clearly improves performance, with the multiGPU setup being the most convenient for these particular problem sizes.

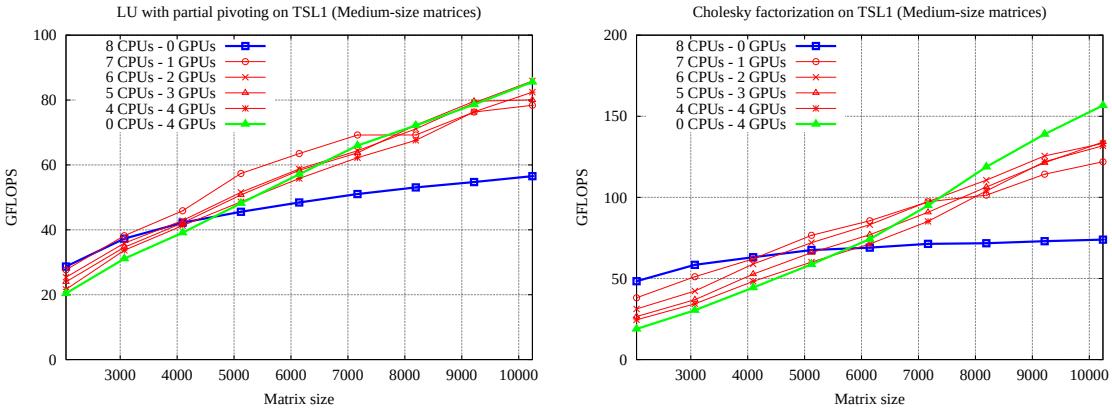
4.4 Advancing critical tasks

Let us analyze in detail the task scheduling of the LU factorization with partial pivoting for a matrix of size $n = 10,240$ with block size $b = 1,024$ using the multiGPU mode on 4 GPUs. Given these dimensions, the Algorithm in Figure 1 partitions the matrix into $s = 10,240/1,024 = 10$ panels. At each iteration $k = 0, 1, \ldots, s-1$, the algorithm proceeds by firstly decomposing the k-th panel of the input matrix ($\mathsf{LU}(k)$); and next updating the trailing submatrix panelwise with respect to the factorization of this panel, which is performed as a sequence of triangular system solves and matrix-matrix updates (tasks $\mathsf{T}(k, j)$ and $\mathsf{G}(k, j)$, respectively, with $j = k + 1, k + 2, \ldots, s - 1$). Hereafter we will refer to the combined application of T and G to the j-th panel as $\textsc{Update}(k, j)$.

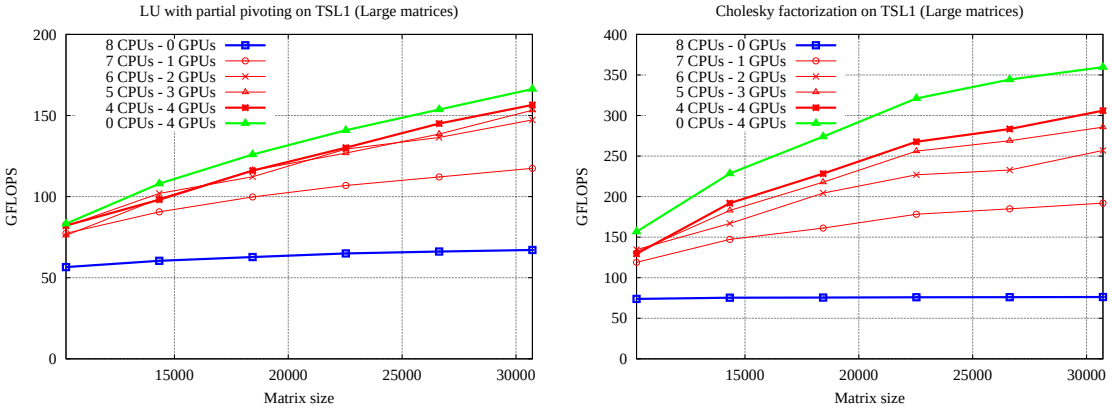
Figure 7 (top) shows a trace for the execution of this LU factorization governed by the original SuperMatrix scheduler, with tracing capabilities provided by `Extrae` and `Paraver` [28]. Note how, in the operation of the original runtime, the factorization $\mathsf{LU}(k + 1)$ does not commence till the update of the full trailing submatrix with respect to the factorization of the previous panel has been completed. An inspection of the order in which tasks are executed there (see the instants marked with numbers 1 to 4 in the trace) reveals that task $\mathsf{LU}(1)$ (execution point 4) does not proceed until the update of the trailing submatrix $\textsc{Update}(0, s-1)$ (execution point 3) is completed.



(a) Small problems.



(b) Medium-size problems.



(c) Large problems.

Figure 6: Performance of LU with partial pivoting and Cholesky factorization on TSL1 using the hybrid runtime scheduler for small problems (top), medium-size problems (middle), and large problems (bottom).

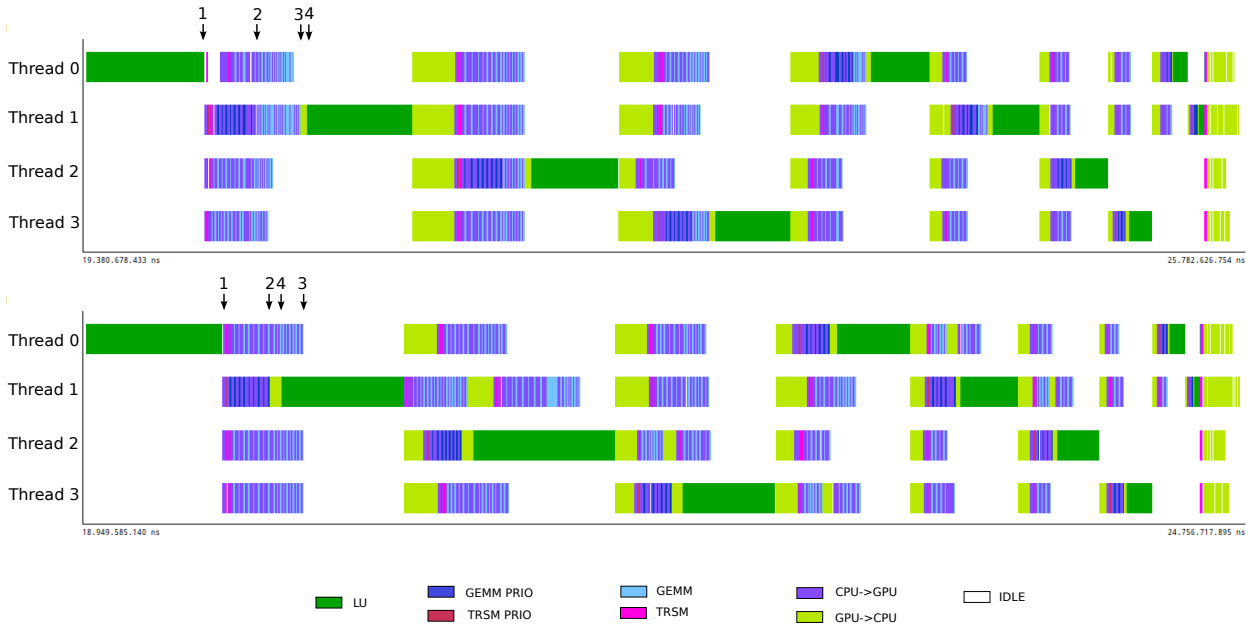


Figure 7: Traces of the execution of the LU factorization with partial pivoting of a matrix of dimension $n = 10,240$, with $b = 1,024$ using 4 GPUs of TSL1, without (top) and with priority tasks (bottom). Selected execution points: 1: End of $\mathsf{LU}(0)$; 2: End of $\textsc{Update}(0,1)$; 3: End of $\textsc{Update}(0,s-1)$; 4: Start of $\mathsf{LU}(1)$.

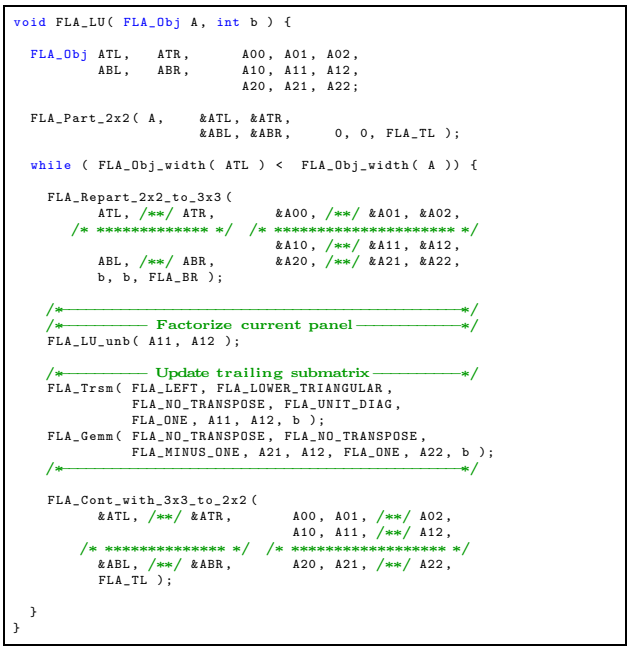
However, the factorization of the current panel and the update of the first panel of the trailing submatrix both lie on the critical path of the TDG (see Figure 2) and, therefore, their execution should proceed as soon as possible. Indeed, task $\mathsf{LU}(k+1)$ could effectively start as soon as task $\textsc{Update}(k,k+1)$ is completed, since the dependencies determine that it is unnecessary to wait for the update of all remaining panels in the trailing submatrix: $\textsc{Update}(k,k+2),\ldots,\textsc{Update}(k,s-1)$. Thus, the goal of our optimization is to enforce a fast execution of those tasks in the critical path, that in practice yields an overlapped execution of $\mathsf{LU}(k+1)$ with $\textsc{Update}(k,k+2),\ldots,\textsc{Update}(k,s-1)$.

To accomplish these goals, tasks in the critical path receive a different treatment in the enhanced version of the SuperMatrix scheduler. Specifically, *i)* critical (or priority) tasks are executed as soon as possible to avoid unnecessary stalls; and *ii)* they are mapped to the fastest computational resource (CPU or GPU) available.

In practice, these restrictions introduce the necessity of *i)* marking certain tasks as *critical*, and *ii)* modifying the scheduler to prioritize the execution of these tasks, mapping them to the fastest suitable execution unit (typically the GPU). In the following, we elaborate on these two requirements.

4.4.1 Identifying priority tasks.

We consider next the blocked algorithm to compute the LU factorization with partial pivoting to illustrate the mechanism to identify priority tasks. An analogous approach can be easily applied for the Cholesky factorization. Remember that the symbolic analysis is performed in SuperMatrix by (automatically) processing the corresponding sequential codes from `libflame` at run time, in order to identify tasks and data dependencies between suboperations, without effectively executing the associated kernels. The actual FLAME/C code used for the blocked LU factorization is given in Figure 8. The code mimics the algorithm in Figure 1 (left), and proceeds by traversing the input matrix A from the top-left corner to the bottom-right one. Similar ideas apply to the blocked Cholesky factorization. While inspecting this procedure, at each iteration the runtime exposes new sub-blocks, adds a new task of type LU to the TDG, and processes the trailing submatrix. The update of this submatrix is performed by invoking routines to perform (a sequence of) triangular system solves (T in routine `FLA_Trsm`) and general matrix-matrix updates (G in routine `FLA_Gemm`), which results in new tasks being added to the TDG; see Figure 9.



```
void FLA_LU( FLA_Obj A, int b ) {

  FLA_Obj ATL,   ATR,      A00, A01, A02,
          ABL,   ABR,      A10, A11, A12,
                           A20, A21, A22;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,     0, 0, FLA_TL );

  while ( FLA_Obj_width( ATL ) <  FLA_Obj_width( A )) {

    FLA_Repart_2x2_to_3x3(
          ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
       /* ************* */  /* ********************* */
                               &A10, /**/ &A11, &A12,
          ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
          b, b, FLA_BR );

    /*-----------------------------------------------*/
    /*---------- Factorize current panel ------------*/
    FLA_LU_unb( A11, A12 );

    /*---------- Update trailing submatrix ----------*/
    FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              FLA_ONE, A11, A12, b );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21, A12, FLA_ONE, A22, b );
    /*-----------------------------------------------*/

    FLA_Cont_with_3x3_to_2x2(
          &ATL, /**/ &ATR,       A00, A01, /**/ A02,
                                 A10, A11, /**/ A12,
        /* ************* */  /* ****************** */
          &ABL, /**/ &ABR,       A20, A21, /**/ A22,
          FLA_TL );

  }
}
```

Figure 8: FLAME code used in the symbolic analysis stage for the LU factorization.

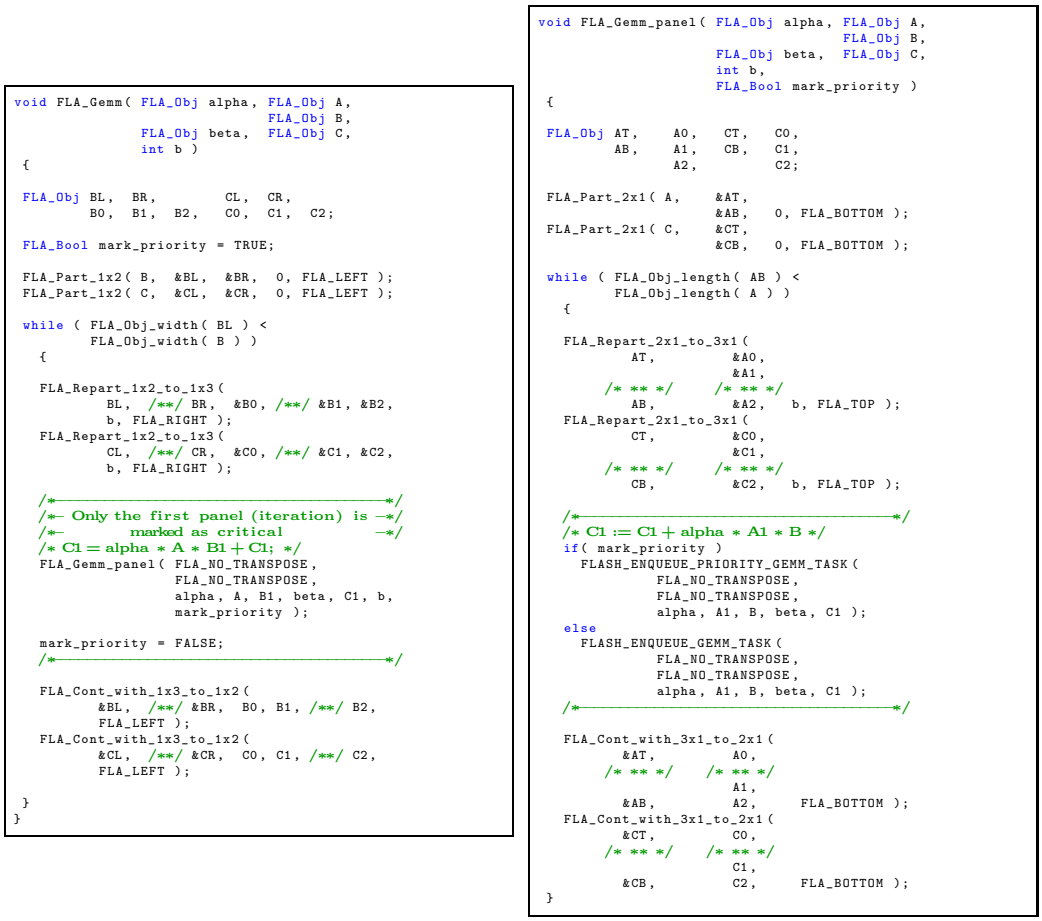
Both `FLA_Trsm` and `FLA_Gemm` proceed by further subdividing the respective operations into smaller T and G sub-operations that update their outputs by panels of columns. In a data-flow run, the specific algorithmic variant chosen for a given operation does not have any effect on the actual execution order as this is only dictated by task dependencies. However, the appropriate choice of variants for the two operations involved in the update of the trailing submatrix makes it easy to identify those tasks that must be marked as critical. In our case, both `FLA_Trsm` and `FLA_Gemm` proceed from left to right updating the trailing submatrix by blocks of columns. Thus, only the tasks that update the first panel lie on the critical path and, therefore, only they must be signaled as priority tasks. Consider the codes in Figure 9 that correspond to the update of the trailing submatrix performed in terms of G. Routine `FLA_Gemm` proceeds by exposing a new panel of columns of C per iteration, and updating it with `FLA_Gemm_panel` by row blocks; the latter performs the actual enqueuing of tasks into the TDG using ad-hoc macros. There, only the first panel update invoked from `FLA_Gemm` will enqueue critical tasks of type G (one per block row). An analogous approach is taken to enqueue/mark critical tasks of type T and the application of permutations in the LU factorization.

4.4.2 Scheduling with priorities.

Once priority tasks are identified and introduced in the TDG, the runtime remains in charge of performing the most adequate action when a ready critical task is encountered. The objective of the scheduler during this stage is two-fold: first, execute priority tasks as *soon* as possible; second, execute them in the *fastest* computational resource that is idle.

The original implementation of SuperMatrix controls a single ready queue containing all tasks with their data dependencies satisfied. (Indeed, when data affinity was in place, there was one queue of ready tasks per thread, containing ready tasks to be run by that thread). In the version enhanced with priorities, the scheduler has been modified to introduce an additional *priority queue* (or one priority queue per thread in case data affinity is used), which is handled as follows:

- After the execution of a task and the needed updates of data dependencies, tasks that become ready are removed from the work queue. If the ready task is critical, it is inserted in the thread *priority queue*; otherwise, it is moved to a *non-priority queue* shared by all threads.
- When a worker thread becomes idle, it polls the queues of ready tasks in order to obtain a new



```
void FLA_Gemm( FLA_Obj alpha, FLA_Obj A,
                              FLA_Obj B,
               FLA_Obj beta,  FLA_Obj C,
               int b )
{

 FLA_Obj BL,  BR,        CL,  CR,
         B0,  B1,  B2,   C0,  C1,  C2;

 FLA_Bool mark_priority = TRUE;

 FLA_Part_1x2( B,  &BL,  &BR,  0, FLA_LEFT );
 FLA_Part_1x2( C,  &CL,  &CR,  0, FLA_LEFT );

 while ( FLA_Obj_width( BL ) <
         FLA_Obj_width( B ) )
   {

   FLA_Repart_1x2_to_1x3(
           BL,  /**/ BR,  &B0, /**/ &B1, &B2,
           b, FLA_RIGHT );
   FLA_Repart_1x2_to_1x3(
           CL,  /**/ CR,  &C0, /**/ &C1, &C2,
           b, FLA_RIGHT );

   /*-------------------------------------------*/
   /*- Only the first panel (iteration) is -*/
   /*-        marked as critical           -*/
   /* C1 = alpha * A * B1 + C1; */
   FLA_Gemm_panel( FLA_NO_TRANSPOSE,
                   FLA_NO_TRANSPOSE,
                   alpha, A, B1, beta, C1, b,
                   mark_priority );

   mark_priority = FALSE;
   /*-------------------------------------------*/

   FLA_Cont_with_1x3_to_1x2(
          &BL,  /**/ &BR,  B0, B1, /**/ B2,
          FLA_LEFT );
   FLA_Cont_with_1x3_to_1x2(
          &CL,  /**/ &CR,  C0, C1, /**/ C2,
          FLA_LEFT );

 }
}
```

```
void FLA_Gemm_panel( FLA_Obj alpha, FLA_Obj A,
                                    FLA_Obj B,
                     FLA_Obj beta,  FLA_Obj C,
                     int b,
                     FLA_Bool mark_priority )
{

FLA_Obj AT,    A0,   CT,   C0,
        AB,    A1,   CB,   C1,
               A2,         C2;

FLA_Part_2x1( A,    &AT,
                    &AB,   0, FLA_BOTTOM );
FLA_Part_2x1( C,    &CT,
                    &CB,   0, FLA_BOTTOM );

while ( FLA_Obj_length( AB ) <
        FLA_Obj_length( A ) )
  {

  FLA_Repart_2x1_to_3x1(
           AT,          &A0,
                        &A1,
       /* ** */     /* ** */
           AB,          &A2,   b, FLA_TOP );
  FLA_Repart_2x1_to_3x1(
           CT,          &C0,
                        &C1,
       /* ** */     /* ** */
           CB,          &C2,   b, FLA_TOP );

  /*-------------------------------------------*/
  /* C1 := C1 + alpha * A1 * B */
  if( mark_priority )
    FLASH_ENQUEUE_PRIORITY_GEMM_TASK(
            FLA_NO_TRANSPOSE,
            FLA_NO_TRANSPOSE,
            alpha, A1, B, beta, C1 );
  else
    FLASH_ENQUEUE_GEMM_TASK(
            FLA_NO_TRANSPOSE,
            FLA_NO_TRANSPOSE,
            alpha, A1, B, beta, C1 );
  /*-------------------------------------------*/

  FLA_Cont_with_3x1_to_2x1(
         &AT,          A0,
       /* ** */    /* ** */
                     A1,
         &AB,        A2,     FLA_BOTTOM );
  FLA_Cont_with_3x1_to_2x1(
         &CT,          C0,
       /* ** */    /* ** */
                     C1,
         &CB,        C2,     FLA_BOTTOM );
}
```

Figure 9: (Left) FLAME code used in the symbolic analysis stage for G. The analysis proceeds from left to right, invoking the corresponding routine for the update of one panel per iteration. (Right) FLAME code used in the symbolic analysis stage for G. At this level, G sub-operations operate on single blocks of the original matrices, enqueuing priority and non-priority tasks to the TDG.

candidate for execution. Consider first the multiGPU mode. If the thread is in control of a GPU, it first checks the corresponding priority queue; if no priority task is available, the shared non-priority queue is polled for a new task. On the other hand, if the thread runs on a CPU core with no GPU attached, only the non-priority queue is polled.

In the multicore mode, the priority queue is always checked first before the non-priority one. This behavior forces the runtime to execute critical tasks as soon as they become ready, doing it on the GPU in the multiGPU mode to ensure a fast execution.

The effect of the enhanced runtime is illustrated in the bottom trace of Figure 7 for the LU factorization with partial pivoting. For this particular operation, an inspection of the order in which tasks are executed exposes that LU(1) (execution point 4) now commences as soon as the update of the first panel of the trailing submatrix has been completed (UPDATE$(0,1)$, execution point 3), effectively overlapping the execution of tasks from both iterations. This reduces the idle time, activating new ready tasks that depend on those of the critical path, and accelerating the parallel execution. A similar effect was observed for the Cholesky factorization.

The performance impact of the improvements introduced in the new runtime is reported in Figures 10 and 11. There, we compare the efficiency of the original version of SuperMatrix with that of the new runtime using the multicore mode and the multiGPU mode of the runtime scheduler. For the LU factorization with partial pivoting, the results show that the performance improvement is especially remarkable for large

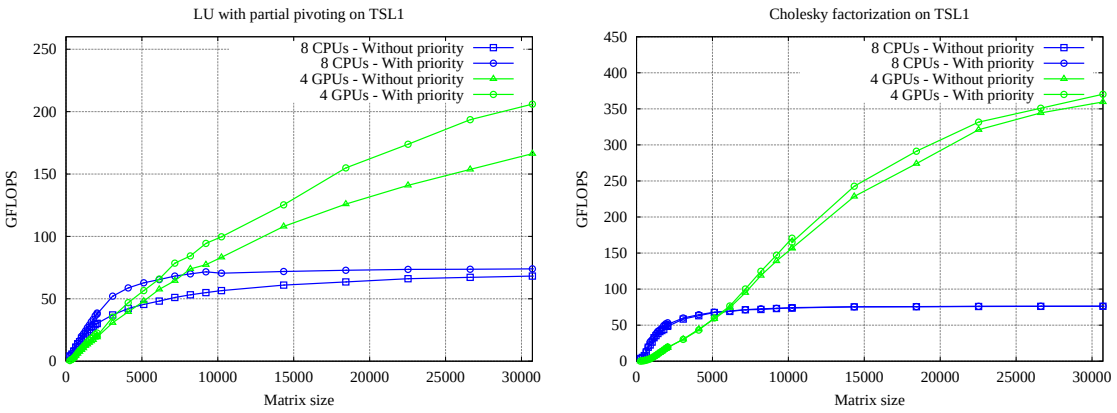


Figure 10: Impact of the use of priority tasks on the performance of the LU factorization with partial pivoting (left-side plot) and Cholesky factorization (right-side plot) on TSL1, using 8 CPU cores (multicore mode) and 4 GPUs (multiGPU mode).

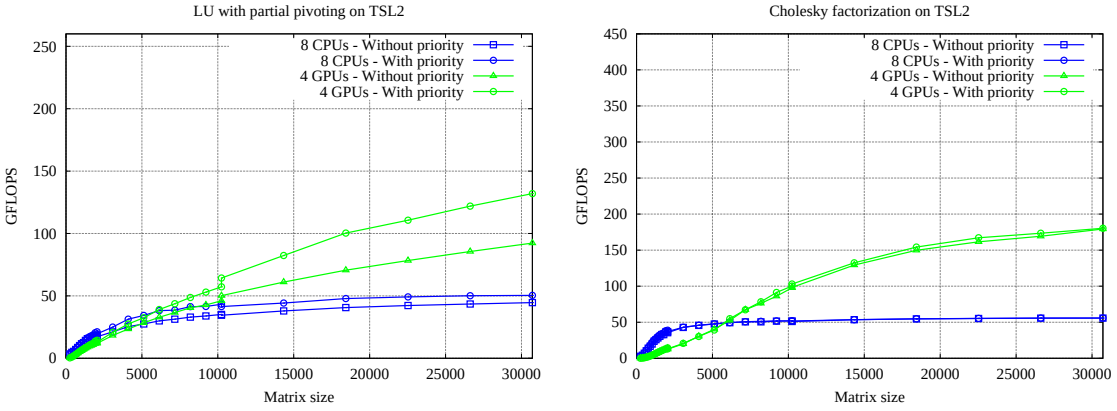


Figure 11: Impact of the use of priority tasks on the performance of the LU factorization with partial pivoting (left-side plot) and Cholesky factorization (right-side plot) on TSL2, using 8 CPU cores (multicore mode) and 4 GPUs (multiGPU mode).

matrices. For the multicore mode, the acceleration varies between 10% and 12%, while for the multiGPU mode, the speedups range from 20% to 25% for the largest tested matrices. For the Cholesky factorization, the improvements attained by the introduction of priority tasks are less important. In the case of the multiGPU mode, the acceleration for the largest tested matrices ranges from 5% to 10%; and no improvements are revealed in this case for the multicore mode. The different impact due to the introduction of priorities is explained by the relative cost of the tasks that lie on the critical path in both cases: for the LU decomposition, the factorization of the current *panel* presents a much higher cost than the factorization of the diagonal *blocks* in the Cholesky decomposition. Thus, advancing those critical tasks (that operate on panels) in the first case can be expected to yield a higher benefit in terms of performance.

5 Energy-Aware Extensions to SuperMatrix

In this section we first present the energy-saving techniques introduced in the runtime, and their practical outcome on the execution of the LU and the Cholesky factorization on TSL1. We then adapt a power model from [29] to the execution of DLA operations on hybrid CPU-GPU platforms, and use the results to put the attained energy savings into perspective.

Table 1: Average power consumption (in Watts) of different actions performed by threads.

	1 core	4 cores
Intel MKL `dgemm`	330	403
Polling	323	367
Blocking	290	293

5.1 Energy-aware runtime

Let us introduce our two energy-saving techniques for the task-parallel execution of FLA/`libflame` routines on hybrid CPU-GPU platforms. In the following we will continue using the LU factorization with partial pivoting to explain the benefits of these techniques, but the ideas and techniques used in the following are also relevant to any other DLA operation.

In order to illustrate these techniques, consider again the bottom trace in Figure 7. The trace was obtained using four control threads (running each on a CPU core) and four GPUs. Furthermore, because of its complexity, LU type tasks were always mapped to the CPU cores while the remaining two types of tasks, T and G, were mapped to the GPUs. Idle time is marked using white color. This corresponds to periods when both CPU cores and GPUs are performing no useful work but waiting for the completion of an event. Inactive periods occur when there are no ready tasks to be executed due to pending dependencies. Moreover, when a GPU is executing a task, the corresponding CPU core is engaged and inoperative, waiting for the GPU to complete the job. The trace certainly shows that these periods occupy a significant fraction of the execution time.

The question that naturally arises is how to leverage these inactive periods to reduce energy consumption. Although Dynamic Voltage and Frequency Scaling (DVFS) [30] can in principle be applied to reduce the power rate during idle periods, our experiments in [31] demonstrated that the savings obtained by reducing the CPU frequency during the execution of compute-intensive DLA operations are small. In particular, Table 1 shows the average power dissipation rates incurred by one and four cores of TSL1 repeatedly executing a matrix-matrix product ("Intel MKL `dgemm`"), performing a busy-wait ("Polling"), and blocked ("Blocking"). Interestingly, depending on the number of active cores, 33-74 Watts can be approximately saved by avoiding polling.

The results of this experiment justify the following two techniques that we introduce in SuperMatrix in order to replace busy-waits by a blocking power-friendly state during the execution of the DLA codes when possible.

5.1.1 Avoid polling when there are no ready tasks.

(Energy-Aware technique 1—EA1) In SuperMatrix, when a CPU thread finishes the execution of a task (and updates the corresponding dependencies), it continuously checks the ready queue in order to request more work, performing an active polling and, therefore, wasting energy in case are no ready tasks. The goal of our first energy-aware technique is to avoid this situation, by using instead a power-friendly blocking-wait (idle-wait). For this purpose, we introduced POSIX semaphores into the runtime to control the activity of "idle" threads. Specifically, when a thread that is polling the ready queue for a new task finds it empty (i.e., there are no ready tasks to be executed at that moment), it blocks itself by calling to the `sem_wait()` routine. This method requires a complementary mechanism to wake up blocked threads. In particular, when an active CPU thread finishes the execution of a task and updates the dependencies within them, in case this requires moving c tasks from the work queue to the ready queue, this thread will also wake up c threads by calling to the system call `sem_post()`. This mechanisms ensures that there will be one active thread per task in the ready queue, minimizing the impact of delays while simultaneously avoiding potential deadlocks.

5.1.2 Avoid polling when waiting for the GPU.

(Energy-Aware technique 2—EA2) When a CPU thread finds a task of type T or G, to be executed on the attached GPU, it automatically invokes the corresponding CUBLAS kernel. Notwithstanding, because of the asynchronous nature of the GPU kernels in NVIDIA CUBLAS, the calling CPU thread does not block,

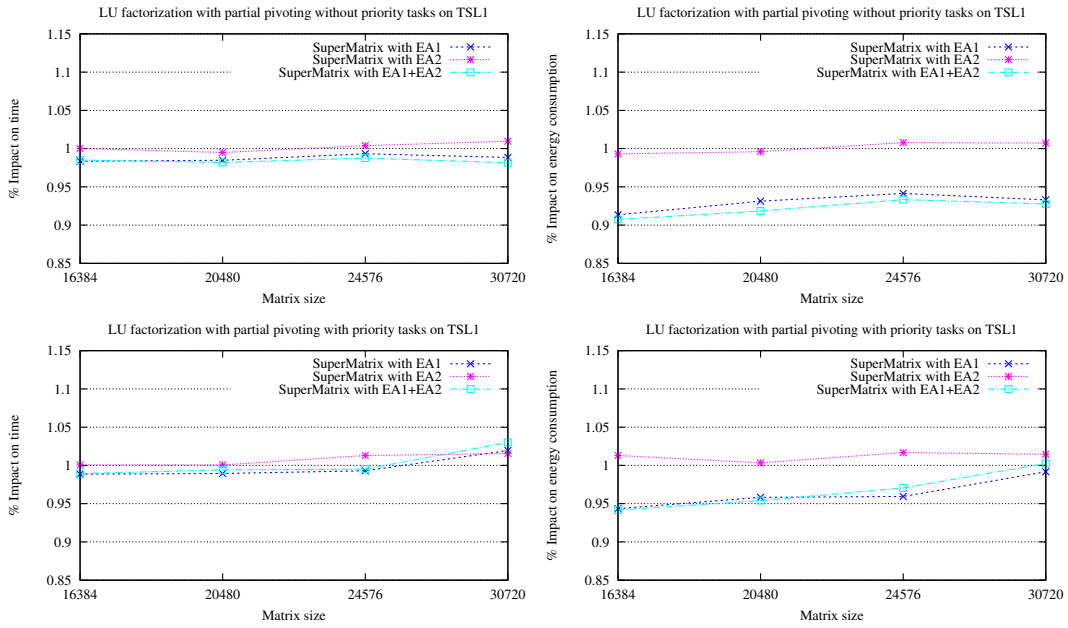


Figure 12: Impact on time and energy of the energy-aware techniques of the LU factorization with partial pivoting without (top plots) and with priority tasks (bottom plots) on TSL1.

and so it queries the ready queue to retrieve a new task. At this point, if the CPU thread dequeues a second task to be run in GPU, it tries to execute it, invoking the appropriate CUBLAS kernel. Consequently, since a GPU is not able to execute more than one kernel simultaneously in the SuperMatrix runtime, the thread commences a busy-wait (polling) until the first kernel finishes its execution on the GPU.

To avoid this behaviour we adapted the runtime so that routine `cudaSetDeviceFlags` is invoked with `cudadeviceBlockingSync` parameter set. This parameter blocks the CPU thread on a synchronization point to wait for the device to finish its work. We also added the correspondent synchronization primitive after CUBLAS calls performed by `libflame`. Routine `cudaSetDeviceFlags` allows to specify the behavior of the active CPU thread when it executes device code. To achieve this, each thread calls this routine before the CUDA runtime is initialized. Afterwards, synchronizations are carried out using the primitive `cudaThreadSynchronize` in order to suspend the execution of the calling thread until the device finishes its work, hence blocking the core and avoiding the potential energy-wasting state.

Figures 12 and 13 report the impact on the execution time and on the energy consumption when the energy-aware version of SuperMatrix without and with priority tasks is employed to execute the LU factorization with partial pivoting on TSL1 and TSL2, respectively. Figures 14 and 15 report the analogous information for the Cholesky decomposition. The combination of both techniques is referred to as "EA1+EA2". These results show a variety of energy gains, from close to 10% in some cases to even a waste of energy in a couple of cases, depending on the DLA operation, matrix dimension, and technique. In the following section we relate these results with the actual consumption of dynamic power and the length of idle periods.

The behavior of the power-aware runtime for the LU factorization without and with task priorities is specially interesting. The reduction in execution time when task priorities are used yields an important reduction in energy consumption (compare right plots in Figure 12 and 13); however, this improvement in execution time is mainly motivated by a reduction in the amount of idle periods in the parallel execution. As our power-aware techniques exploit idle periods, the expected improvements from the application of these mechanisms are less significant as idle time decreases. That explains why the percentage of reduction in energy consumption is smaller when priorities are applied (less than 6%) than when priorities are not used (up to 9%).

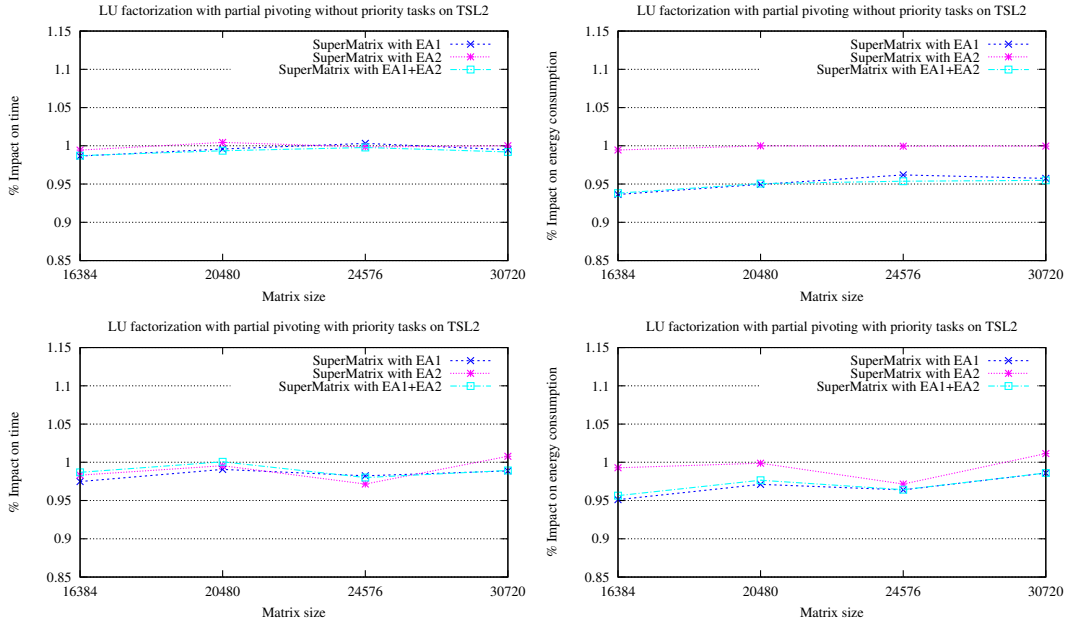


Figure 13: Impact on time and energy of the energy-aware techniques of the LU factorization with partial pivoting without (top plots) and with priority tasks (bottom plots) on TSL2.

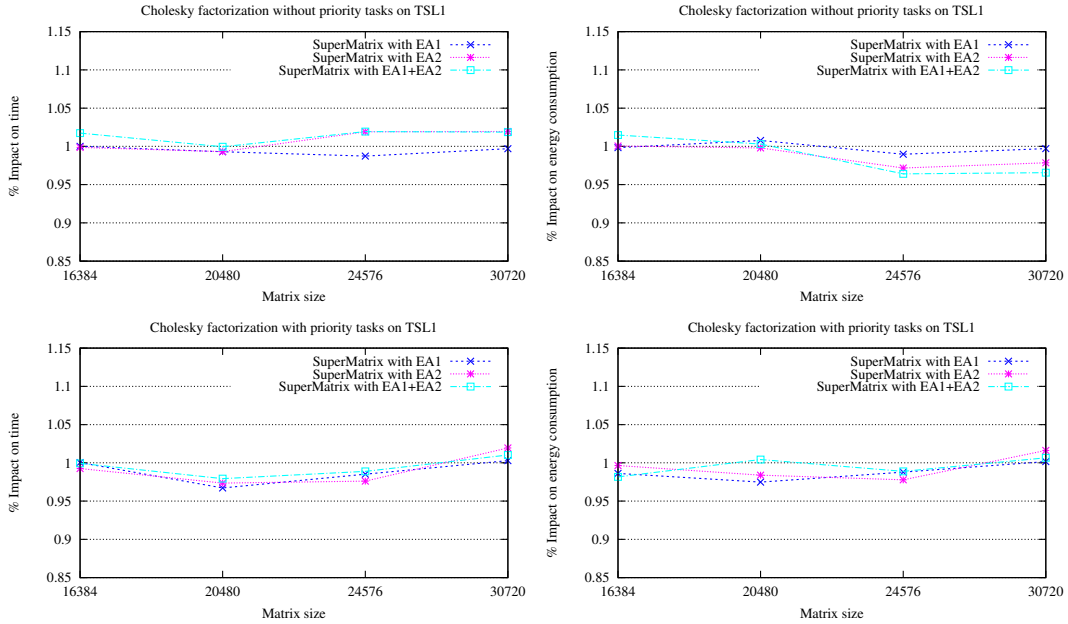


Figure 14: Impact on time and energy of the energy-aware techniques of the Cholesky factorization without (top plots) and with priority tasks (bottom plots) on TSL1.

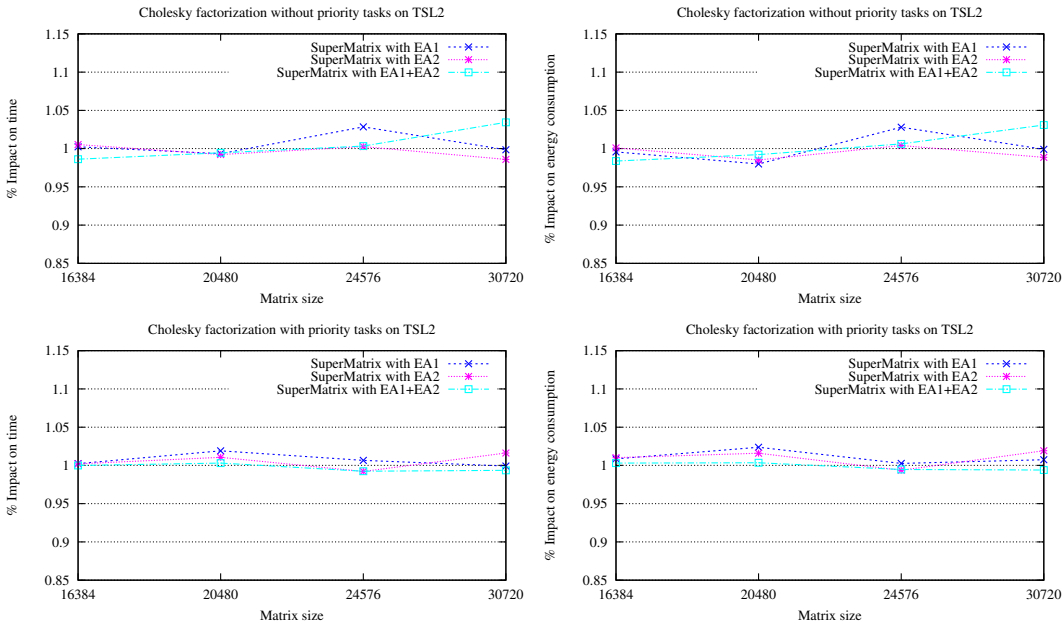


Figure 15: Impact on time and energy of the energy-aware techniques of the Cholesky factorization without (top plots) and with priority tasks (bottom plots) on TSL2.

5.2 A model of power consumption

We next review a power model initially proposed in [29], and customize it to assess the quality of the energy-savings observed in the previous experiments. Given that the energy savings due to EA1 and EA2 techniques are produced in the CPU cores, we only dissect the power consumption in the host. The proposed model can eventually be incorporated into the runtime to dynamically determine the optimal number of computational resources to employ for the execution of a DLA operation. Since there exists different configurations of resources that result in equivalent or nearly equivalent performance, the *hybrid* SuperMatrix runtime can make use of the model to automatically select the most energy-efficient configuration.

Consider the following simple model, borrowed from [32]:

$$P = P^C + P^Y = P^S + P^D + P^Y,$$

whereby we decompose the total power consumption P as a sum of P^C, which is the power dissipated by the CPU, and P^Y, which is the power consumption due to the remaining components (system power corresponding, e.g, to RAM). We decompose P^C in turn into the static power (mainly due to leakage) and the dynamic power, P^S and P^D, respectively.

We assume that P^S and P^Y remain constant during the execution of the algorithm. In practice, starting from an idle platform, P^S grows with the system temperature and activity up to a maximum [32]. In order to avoid this effect, all our tests were performed on a "hot" system representing a platform where always exists a continuous compute-intensive workload to execute. The dynamic power is a function of activity and therefore time, and so it is also the total power consumption:

$$P(t) = P^S + P^D(t) + P^Y\,. \tag{1}$$

Finally, the energy consumption is obtained by integrating the power dissipation over the total execution time T:

$$E \;=\; \int_{t=0}^{T} P(t)\;dt = (P^Y + P^S)\cdot T + \int_{t=0}^{T} P^D(t)\;dt. \tag{2}$$

We next employ a practical approach to determine the parameters necessary to assemble, refine and customize the models in (1) and (2) for the particular case of the execution of the task-parallel LU factorization with partial pivoting on TSL1. The model for TSL2 and/or the Cholesky factorization is obtained following an analogous procedure.

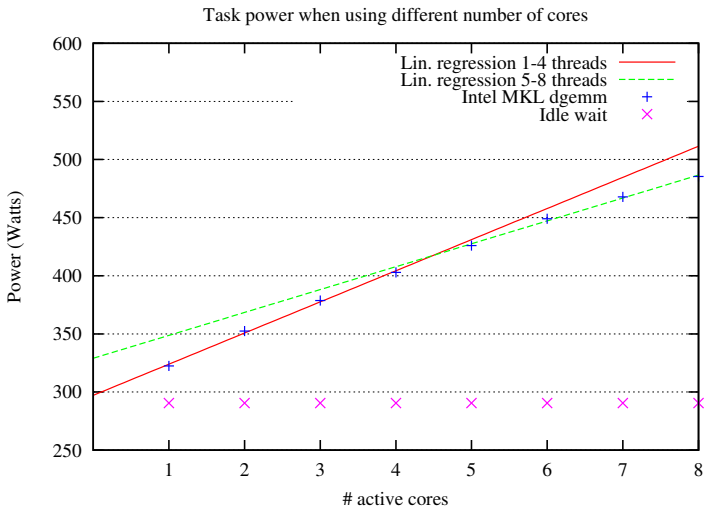


Figure 16: Power dissipated as a function of number of active cores on TSL1.

5.2.1 System and static power.

Let us first estimate the system and static parameters of the power model (1). Direct measurement using our external PDU with the platform completely idle revealed a power dissipation of 290.47 Watts, which can be taken as an approximation for P^Y. On the other hand, Figure 16 reports the power consumption executing a matrix-matrix product (routine `dgemm` from Intel MKL) with square operands of size 1,024. We vary the number of CPU threads/cores of TSL1 from 1 to 4 when running on the same socket, and from 5 to 8 when running on the two sockets. Applying a linear regression to adjust the total power, two linear models are obtained: $P_{\texttt{dgemm}}(c) = \alpha + \beta \cdot c = 297.16 + 26.78 \cdot c$ Watts for the case of 1–4 threads, and $P'_{\texttt{dgemm}}(c) = \alpha' + \beta' \cdot c = 329.02 + 19.69 \cdot c$ Watts for the case of 5–8 threads. Therefore, the static power dissipated by the system can be approximated as $P^{S1} \approx \alpha - P^Y = 6.68$ Watts when there is only one socket active, and $P^{S2} \approx \alpha' - P^Y = 38.55$ Watts otherwise.

5.2.2 Dynamic power.

The characterization of the dynamic power dissipated by a task-parallel DLA operation is more challenging. In the case of the LU factorization with partial pivoting, we have to estimate the power dissipated during the execution of each one of the three kernels (task types) that appear —LU, T and G— as well as the consumption due to data transfers.

The previous experiment revealed that the dynamic power increases linearly with the number of threads mapped to a single socket that execute a certain task (the matrix-matrix product in that case). Besides, there we could also observe that the use of cores from two sockets changes the arguments which define this linear function. To account for this difference during the estimation of each one of the kernels, we performed an experiment where one single thread continuously invokes one type of kernel, e.g. G, till the total power stabilizes; we then sampled this value, say P_{G}, and set $P_{\mathsf{G}}^{D1} = P_{\mathsf{G}} - P^S - P^Y = P_{\mathsf{G}} - 297.16$ Watts. We next repeated the experiment, with two concurrent CPU threads invoking the same kernel on different sockets of the target platform and, when the power stabilized, we sampled the value, P'_{G}, and set $P_{\mathsf{G}}^{D2} = (P'_{\mathsf{G}} - 329.02)/2$ Watts. The estimations of the dynamic power for each kernel type and number of active sockets are collected in Table 2. These values were obtained employing operands of square dimension 1,024 in all cases.

5.2.3 Power dissipation and energy consumption of DLA operations.

Consider finally the task-parallel execution of a DLA operation, with the algorithm decomposed into r different types of tasks, with r_j instances of task type j. The total power dissipation of the algorithm, at an instant of time t, is given by the composition of the system power, the static power, and the dynamic power

Table 2: Dynamic power (in Watts) of the task types involved in the LU factorization with partial pivoting, estimated when placing 1 and 2 threads in different sockets of TSL1.

Task	LU	T	G	CPU→GPU	GPU→CPU
P^{D1}	27.92	36.32	26.03	50.33	49.63
P^{D2}	14.53	16.03	19.43	–	–

of all tasks being executed at that instant:

$$\begin{aligned} P(t) &= P^Y + P^S(t) + P^D(t) \\ &= P^Y + P^{S1} M(t) + P^{S2} M'(t) + \sum_{j=1}^{r} \sum_{i=1}^{r_j} \left(P_j^{D1} M(t) + P^{D2} M'(t) \right) N_{i,j}(t). \end{aligned}$$

Here, $M(t)$ is a function that, at time t, returns 1 in case there are active threads running in a single socket only or equals 0 otherwise; similarly, $M'(t) = 1$ if at time t there exist active threads distributed between the two sockets or equals 0 otherwise; and $N_{i,j}(t) = 1$ if the task (i, j) is being executed at time t or equals 0 otherwise.

The energy consumption of the algorithm is a function of the time spent in each type of task:

$$\begin{aligned} E &= \int_{t=0}^{T} P(t)\ dt \\ &= P^Y\, T + P^{S1}\ S + P^{S2}\ S' + \sum_{j=1}^{r} \sum_{i=1}^{r_j} P_j^{D1}\ T_j\ \gamma_{i,j} + P_j^{D2}\ T_j\ (1 - \gamma_{i,j}) \end{aligned}$$

where S and S' are the periods of time when there is a single socket and two sockets active, respectively; T_j is the execution time of a task of type j; P_j^{D1}, P_j^{D2} are the dynamic powers given in Table 2, with j referring to the different types of tasks (columns of the table); and $\gamma_{i,j}$ is the ratio of time that there is one socket active during the execution of task (i, j). Note that, to accurately determine the energy consumption, we need a detailed *trace* of the algorithm execution.

This energy model can then be used to evaluate the quality of the gains attained by the energy savings techniques. Consider, e.g., the execution of the LU factorization with partial pivoting, for a matrix of dimension $n = 20,480$ with block size $b = 1,024$, using the multiGPU mode and 4 GPUs. The total execution time employing the runtime enhanced with priorities is 39.0 seconds, of which CPU cores are inactive during 50.9%. On the other hand, the total energy consumption is 13,969.9 Joules, with about 82.9% corresponding to the sum of system and static power dissipation. Thus, the highest savings that we could expect by leveraging those periods during which CPU cores are idle is approximately $0.509 \cdot (1 - 0.829) \approx 7.6\%$, which is consistent with the savings reported in Figure 12 for that particular problem size.

6 Concluding Remarks and Future Work

The HPC community is currently well aware that reducing the energy drawn from compute-intensive applications is a concern almost in equal terms with the conventional quest for high performance. While these two factors seem in principle orthogonal, our insights shown throughout this paper reveal mutual implications that addressed together yield relevant synergies.

In particular, we have introduced significant enhancements in the SuperMatrix runtime scheduler to improve performance. The first one consists of a new *hybrid execution* mode that renders a better use of the available hardware resources in modern CPU/GPU platforms. The second enhancement introduces *priority tasks* that, adequately managed, yields a reduction of the idle time caused by dependencies in the critical path. The experimental results show a speedup around 20–25% for the LU factorization with partial pivoting and 5–10% for the Cholesky factorization in the multiGPU configuration. The same techniques carry beyond other DLA routines included in `libflame` and they can also be incorporated into other general-purpose runtime schedulers that exploit task-parallelism.

Furthermore, we also proposed two techniques which aim at reducing power dissipation at idle times during the parallel executions on hybrid CPU/GPU platforms, with a minimal (or even null) impact on the execution time. The techniques have also been integrated into the runtime scheduler and are transparent to the programmer.

Finally, we have elaborated a *power model* that can pass valuable information to the scheduler, so that at run time it can make decisions on the best configuration from the energy perspective. The model also gives the ability to extract theoretical bounds on the power savings for specific operations and it is general enough to be applicable in other task-parallel environments or runtime schedulers.

As part of future work, we intend to explore new scheduling algorithms that combine data locality with energy-saving techniques.

Acknowledgments

This research was supported by project CICYT TIN2011-23283 and FEDER, and by the EU-FET grant "EXA2GREEN" 318793. Francisco D. Igual was supported by project TIN2012-32180.

References

[1] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. 2.3.1 edn. August 2009.

[2] Project home page for OpenCL - the open standard for parallel programming of heterogeneous systems. project home page. `http://www.khronos.org/opencl/`.

[3] The Green500 list 2010. Available at `http://www.green500.org`.

[4] The top500 list 2010. Available at `http://www.top500.org`.

[5] OmpSs project home page. `http://pm.bsc.es/ompss/`.

[6] StarPU project home page. `http://runtime.bordeaux.inria.fr/StarPU/`.

[7] Mentat project. `http://www.cs.virginia.edu/~mentat/`.

[8] Harmony project home page. `http://code.google.com/p/harmonyruntime/`.

[9] Cilk project. `http://supertech.csail.mit.edu/cilk/`.

[10] Badia RM, Herrero JR, Labarta J, Pérez JM, Quintana-Ortí ES, Quintana-Ortí G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience* 2009; **21**(18):2438–2456.

[11] PLASMA project home page. `http://icl.cs.utk.edu/plasma/`.

[12] FLAME project home page. `http://www.cs.utexas.edu/users/flame/`.

[13] Borkar S, Chien AA. The future of microprocessors. *Communications of the ACM* 2011; **54**(5):67–77.

[14] Esmaeilzadeh H, Blem E, St Amant R, Sankaralingam K, Burger D. Dark silicon and the end of multicore scaling. *Proc. 38th Annual Int. Symp. Computer architecture*, ISCA '11, 2011; 365–376.

[15] Duranton M, *et al.* The HiPEAC vision for advanced computing in horizon 2020 2013. URL `http://www.hipeac.net/roadmap`.

[16] Zee FGV. `libflame`. the complete reference 2008. `http://www.cs.utexas.edu/users/flame`.

[17] Quintana-Ortí G, Quintana-Ortí ES, van de Geijn RA, Zee FGV, Chan E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software* 2009; **36**(3):14:1–14:26.

[18] Quintana-Ortí G, Igual FD, Quintana-Ortí ES, van de Geijn R. Solving dense linear algebra problems on platforms with multiple hardware accelerators. *PPoPP '09: The 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, USA, 2009; 121–129.

[19] Igual FD, Quintana-Ortí G, van de Geijn R. Scheduling algorithms-by-blocks on small clusters. *Concurrency and Computation: Practice and Experience* 2012; doi:10.1002/cpe.2842. URL `http://dx.doi.org/10.1002/cpe.2842`.

[20] Alonso P, Dolz MF, Igual FD, Quintana-Ortí ES, Mayo R. Runtime scheduling of the LU factorization: Performance and energy. *Energy Efficiency in Large Scale Distributed Systems (EELSDS 2013). Lecture Notes in Computer Science*, vol. 8046, 2013; 153–167.

[21] Bientinesi P, Gunnels JA, Myers ME, Quintana-Ortí ES, van de Geijn RA. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software* 2005; **31**(1):1–26.

[22] Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach*. 5th edn., Morgan Kaufmann Pub.: San Francisco, 2012.

[23] Alonso P, Badia RM, Labarta J, Barreda M, Dolz MF, Mayo R, Quintana-Ortí ES, Reyes R. Tools for power-energy modelling and analysis of parallel scientific applications. *ICPP*, IEEE Computer Society, 2012; 420–429.

[24] Barrachina S, Castillo M, Igual FD, Mayo R, Quintana-Ortí ES, Quintana-Ortí G. Exploiting the capabilities of modern GPUs for dense matrix computations. *Conc. and Comp.: Pract. and Exper.* 2009; **21**(18):2457–2477.

[25] Chan E, van de Geijn R, Chapman A. Managing the complexity of lookahead for lu factorization with pivoting. *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, ACM: New York, NY, USA, 2010; 200–208, doi:10.1145/1810479.1810520. URL `http://doi.acm.org/10.1145/1810479.1810520`.

[26] Igual FD, Chan E, Quintana-Ortí ES, Quintana-Ortí G, van de Geijn RA, Zee FGV. The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing* 2012; **72**(9):1134 – 1143, doi:10.1016/j.jpdc.2011.10.014. URL `http://www.sciencedirect.com/science/article/pii/S0743731511002139`, accelerators for High-Performance Computing.

[27] Pérez JM, Bellens P, Badía RM, Labarta J. CellSs: Programming the Cell/B.E. made easier 2007.

[28] Paraver project. `http://www.cepba.upc.es/paraver`.

[29] Alonso P, Dolz MF, Mayo R, Quintana-Ortí ES. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *Computer Science - Research and Development* 2012; Available online.

[30] Elnozahy E, Kistler M, Rajamony R. Energy-efficient server clusters. *Power-Aware Computer Systems Second International Workshop, PACS 2002, Lecture Notes in Computer Science (LNCS)*, vol. 2325, Springer-Verlag: Cambridge, MA, USA, 2003; 179–197.

[31] Alonso P, Dolz MF, Igual FD, Mayo R, Quintana-Ortí ES. Saving energy in the lu factorization with partial pivoting on multi-core processors. *PDP*, Stotzka R, Schiffers M, Cotronis Y (eds.), IEEE, 2012; 353–358.

[32] AnandTech Forums. Power-consumption scaling with clockspeed and Vcc for the i7-2600K. `http://forums.anandtech.com/showthread.php?t=2195927` 2011.