TACC Technical Report TR-12-04

Dense Matrix Computation on a Heterogenous Architecture: A Block Synchronous Approach

Kyungjoo Kim* Victor Eijkhout* Robert A. van de Geijn*

August 5, 2012

This technical report is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that anyone wanting to cite or reproduce it ascertains that no published version in journal or proceedings exists.

Permission to copy this report is granted for electronic viewing and single-copy printing. Permissible uses are research and browsing. Specifically prohibited are *sales* of any copy, whether electronic or hardcopy, for any purpose. Also prohibited is copying, excerpting or extensive quoting of any report in another work without the written permission of one of the report's authors.

The University of Texas at Austin and the Texas Advanced Computing Center make no warranty, express or implied, nor assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed.

* The University of Texas at Austin, Austin, TX 78712

Abstract

We present a strategy for efficient use of all components of a heterogenous compute node of a typical current generation cluster. Such nodes often comprise multiple sockets with a multicore processor per socket and one or more accelerators, possibly from different generations and/or types. Our strategy differs from schedulers such as Quark or SuperMatrix in that it does not rely on a Directed Acyclic Graph, but rather uses a bulk-synchronous model. Also, it uses dynamic task division rather than aggregation to deal with the heterogeneous components of a node. Practical experiments show the merits of our approach.

Keywords

Multicore, MultiGPU, Dense Linear Algebra, Algorithms-by-Blocks, Matrix Factorization, Heterogeneous Architectures, BLAS

1 Introduction

Heterogeneous architectures consisting of, for example, multicore processors with attached accelerators are becoming commonplace, especially in nodes for High Performance Computing (HPC) platforms. How to harness these resources is now a topic of research.

For dense matrix computations, a convenient solution has been to view matrices as consisting of blocks (submatrices) and to formulate computations as algorithms-by-blocks (called tile algorithms by some). Blocks then become units of data, operations with blocks become tasks (units of computation), and executing an algorithm-by-blocks generates a Directed Acyclic Graph (DAG) of tasks and dependencies that is then dynamically scheduled to compute resources.

The problem with this solution is that when the computational power of the available resources greatly differs, the homogeneous blocking of the matrix creates a tension between two conflicting issues:

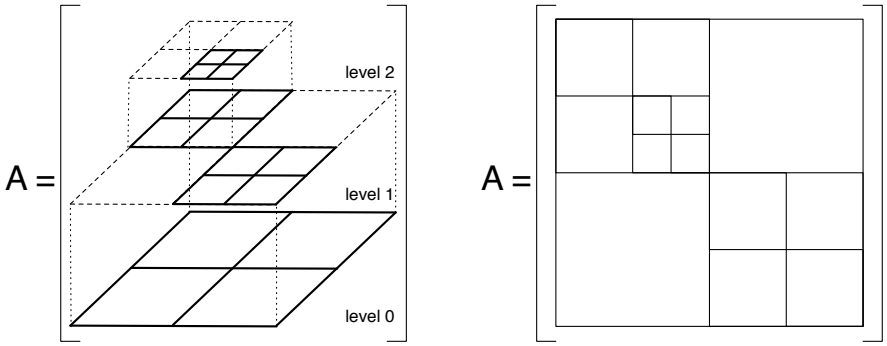
- The block size should be relatively small to be able to assign smaller units of computation to slower devices without holding up faster devices.
- The block size should be relatively large in order to get the best performance from the faster devices.

In previous solutions, the block size has been chosen relatively small, and these smaller tasks have been implicitly or explicitly aggregated to create larger tasks for the faster devices. Still, the individual tasks remain small and achieve suboptimal performance on the fastest devices.

The study described in this paper takes an alternative approach. The matrices are blocked into a size that allows the fastest devices to achieve high performance. Teams of slower devices are created that together have computational power similar to that of the fastest device. Tasks are assigned to these resources (the set of fast devices and teams) as resources become available. This means that when a task is assigned to a team, the matrix blocks associated with that task need to be subdivided into smaller blocks and smaller tasks associates with those smaller blocks scheduled to the members of the team, as illustrated in Fig. 1. If there are more than two types of devices, this may continue recursively. No DAG of tasks is created in our approach; instead, the computation is staged as a sequence of supersteps. Each superstep consists of tasks that can be executed concurrently, in a bulk synchronous fashion.

The approach solves an additional common problem: Often some tasks cannot be performed on the accelerator. For example, the CUBLAS do not include a Choleksy factorization that executes on a GPU[1], which is needed when implementing a larger

1. More recently, the CULA library and Matrix Algebra on GPU and Multicore Architectures



(a) Multi-level matrix resulting from irregular partitioning

(b) Overlayed structure of partitioned matrices

Figure 1: A multi-level matrix (matrix of matrices) is recursively defined with block objects which view a part of the base flat matrix. The left figure shows an example for multi-level partitions layered on the base matrix. Proper index translation is concealed in each block object. The right figure shows its irregular layout of blocks.

Cholesky factorization. In our approach, such tasks are executed by a team of slower devices.

Our results are the following.

- We explore dynamic scheduling to improve device efficiency and load balancing for dense matrix computations.
- The proposed approach can be coded at a high level of abstraction similar to that used by the FLAME project. It is flexible in its treatment of different heterogeneous architectures.
- Performance that is comparable to or better than that of a DAG-based approach can be achieved. Performance is robust for different device configurations. Thus, our approach is a viable alternative to existing packages.

2 Related Works

We briefly discuss prior work related to mapping dense matrix computations to multicore and multiGPU architectures.

(MAGMA) project do provide such higher level functionality. However, this is often available well after a device is first introduced.

Task parallelism on multicore processors. The Basic Linear Algebra Subprograms (BLAS) [10, 7, 8] is an interface for routines that perform basic operations with vectors and dense matrices. Multithreaded implementations explore functional-level parallelism and each function operates in parallel. Its parallel efficiency is often limited because of synchronization that occurs at the end of a BLAS call. Typically, multithreaded BLAS may devote all resources for a single matrix computation regardless of problem sizes and the number of independent problems.

Advanced Dense Linear Algebra (DLA) libraries such as `libflame` [27] and Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [4] achieve efficient task-level parallelism for multicore processors by viewing the computation as a DAG of tasks, to be scheduled to the cores. Matrices are viewed as consisting of blocks (submatrices) that become units of data. Computation is organized as tasks that operate on blocks, yielding algorithm-by-blocks (called tile algorithms by the PLASMA community). These tasks and the data dependencies between them form a DAG. As the dependencies for tasks become satisfied, they are scheduled to the available cores by a runtime. The runtime developed by the FLAME project, called SuperMatrix [5, 6] is another DAG scheduler.

Dense kernels on GPUs. After Nvidia introduced CUDA[16] for general purpose applications, Graphic Processing Units (GPUs) have received substantial attention from the HPC community due to the high efficiency that they can attain via data parallelism (stream processing). Nvidia also offers numerical libraries that are optimized for a single GPU, the CUBLAS [15]. Impressive performance improvements were made to the CUBLAS by adopting Volkov's implementations [29]. As for the newest architecture *Fermi*, CUBLAS uses an improved version of GEMM [14, 23]. The problem with the CUBLAS is that they only target operations executed entirely on a single GPU, ignoring the host processor or other attached devices. The MAGMA [1] project also provides BLAS and Linear Algebra PACKage (LAPACK) functionality on GPUs.

Hybrid computing accelerated by GPUs. It was quickly observed that DAG scheduling of tasks can just as easily target multiple GPUs and/or heterogeneous platforms. When a task can be performed on a GPU, it can be scheduled to a GPU. When it must be performed on a conventional core, it can be scheduled to a conventional core. The runtime system, SuperMatrix for the `libflame` library [19] and more recently the Quark scheduler for PLASMA, can manage the resources [24]. Both DLA libraries extend their task scheduling algorithms that are originally developed for homogeneous multicore architectures to heterogeneous computing, However, both approaches are limited as they use uniform blocks in matrices, which causes inefficiency on differ-

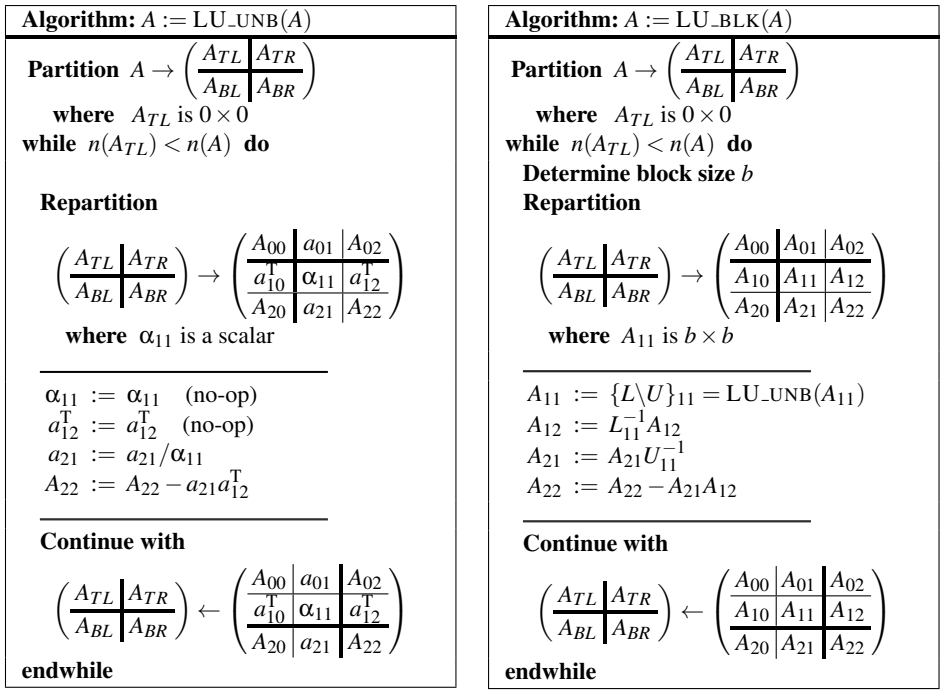


Figure 2: Unblocked and blocked algorithms (left and right) for the LU factorization without pivoting. Here, $n(B)$ stands for the number of columns of B.

ent devices. Song et al. [22] uses 1D rectangular partitions and statically distributes corresponding blocks to GPUs and a multicore processor. Providing non-uniform 1D partitions on matrices, the approach can efficiently exploit the asymmetric performance of different devices. However, their approach adds complexity by requiring an auto-tuning procedure for workload balancing and it is difficult to obtain higher efficiency using 1D static partitions on complex heterogeneous systems that are more than two types of devices.

3 Motivating Example

In this section, we use the examples of LU factorization without pivoting to motivate the proposed approach.

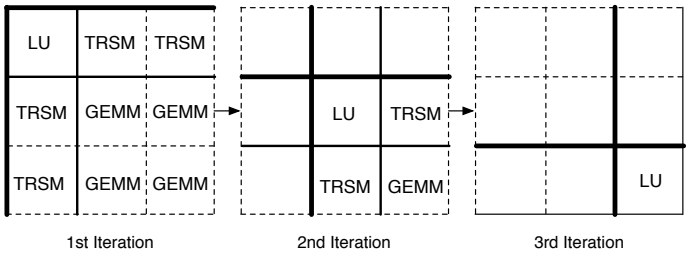


Figure 3: Identified tasks during block LU factorization without pivoting.

3.1 Blocked algorithms

Given a matrix $A \in \mathbb{R}^{n \times n}$, the LU factorization computes L and U such that $A = LU$, where L is $n \times n$ unit lower triangular and U is $n \times n$ upper triangular. In Fig. 2, so-called right-looking unblocked and blocked algorithms are given for computing L and U using the Formal Linear Algebra Methods Environment (FLAME) notation [18, 26] that hides indexing details. The blocked algorithm casts most computation in terms of matrix-matrix multiplication, which can attain high performance.

3.2 Algorithms-by-blocks

A number of projects have suggested that matrices should be viewed as collections of blocks (submatrices), possibly hierarchically [9, 11, 25]. Algorithms such as LU without pivoting can then be formulated as *algorithm-by-blocks*, where blocks are units of data and computation with blocks are tasks [4, 20]. For example, in Fig. 3 the matrix is partitioned into an $N \times N$ matrix of blocks:

$$A = \begin{pmatrix} A^{(0,0)} & A^{(0,1)} & \cdots & A^{(0,N-1)} \\ A^{(1,0)} & A^{(1,1)} & \cdots & A^{(1,N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(N-1,0)} & A^{(N-1,1)} & \cdots & A^{(N-1,N-1)} \end{pmatrix}$$

In the first iteration,

- $A^{(0,0)}$ is factored,
- for $i = 1, \ldots, N-1$ block $A^{(i,0)}$ is overwritten by $A^{(i,0)}U^{(0,0)-1}$; this triangular solve with multiple right-hand sides is done with the BLAS routine TRSM;
- for $k = 1, \ldots N-1$ block $A^{(0,k)}$ is overwritten by $L^{(0,0)-1}A^{(0,k)}$; this is a triangular solve with multiple right-hand sides, executed by TRSM, and
- for $i,k = 1, \ldots, N-1$ blocks $A^{(i,k)}$ are overwritten by $A^{(i,k)} - A^{(i,0)}A^{(0,k)}$; requires a matrix-matrix multiplication using the BLAS routine GEMM.

This and subsequent iterations are illustrated in Fig. 3.

3.3 Scheduling DAGs

In a currently popular approach, parallelism is extracted from algorithm-by-blocks by organizing the tasks and dependencies into a DAG, and then scheduling the tasks to threads as dependencies are satisfied. Optimizations include scheduling the tasks out-of-order to improve temporal and/or spacial locality.

When targeting heterogeneous architectures, *e.g*, multicore processors with multiple (GPU) accelerators, this poses challenges:

- The typical accelerator attains much greater performance than a traditional CPU core.
- Not all operations with blocks may have been implemented on the accelerators. For example, the LU without pivoting is not supported as part of the CUBLAS for Nvidia GPUs.

The second problem is overcome by executing that task on one or more cores. The first problem creates a tension between wanting to keep matrix blocks large, since then tasks achieve higher efficiency on the accelerators, and wanting to keep them small since otherwise the tasks that must be (or are chosen to be) executed on one of more cores may become a bottleneck.

3.4 Bulk synchronous scheduling

Earlier work on task scheduling mostly focused on how to sort tasks and dispatch these to a number of threads in an attempt to minimize idle time on threads. Tasks are typically created by a uniform partitioning of the matrix and fast devices such as GPUs are accommodated by task aggregation of these smaller tasks. However, fast devices typically have a larger optimal block size so this approach meaning that the most potent resource is suboptimally utilized. We solve this problem by taking the opposite approach: we block the data and tasks with a block size that tailors to the fastest devices, and dynamically subdivide this for the slower devices. Another way of thinking of this is that instead of aggregating tasks for the fastest devices, we aggregate slower devices for the coarse block size.

In this section, we illustrate this with a concrete example, where we let our target architecture consist of a quad-core processor and a GPU. The general case is discussed in Section 5. Simplified performance profiles for the both devices are depicted in Fig. 4. In this example, the performance of the single core unit approaches its peak at the problem size b and stay relatively flat whereas the GPU starts to show peak performance at βb, with $\beta = 2$ in this example. Additionally, we state that the GPU is three times faster than the multicore processor.

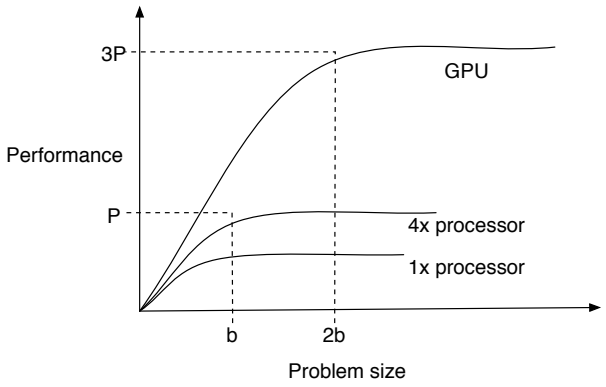
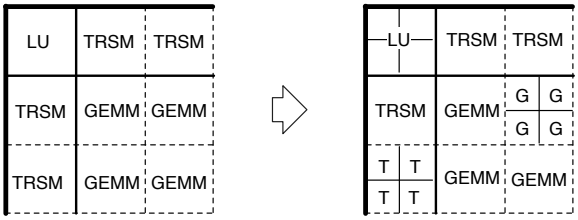


Figure 4: Performance profiles of an example multicore processor and a single GPU.

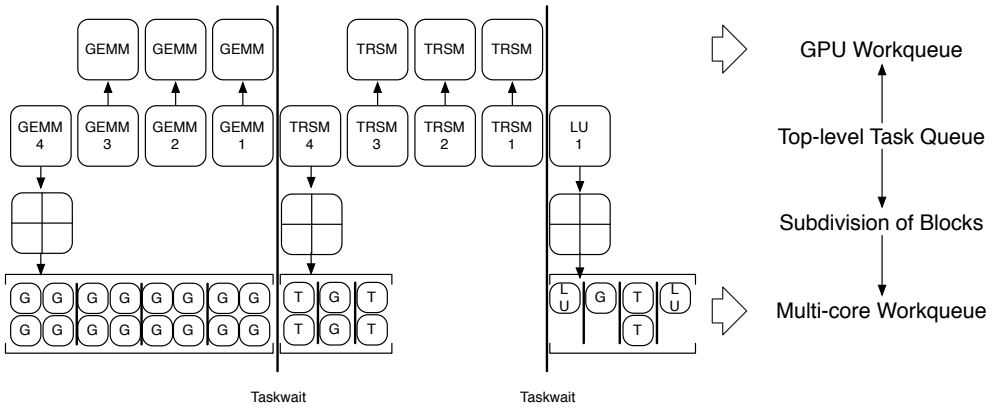
Determining block sizes. In the hybrid context, it is unlikely that a single algorithmic block size is optimal for all heterogeneous computing resources, and an appropriate block size should be selected for each device. A block size is usually determined by taking account various computer architectural features, such as memory hierarchy, the number of processing units, and performance profiles. Higher performance in a GPU naturally leads to use a larger block size while a smaller block size is preferred for a multicore processor to improve parallel efficiency and workload balance. In this example, the multicore processor prefers a block size of b whereas a GPU prefers $2b$. Block sizes can be empirically found by examining the performance of a single device.

Heterogeneous task scheduling. We now return to the LU factorization without pivoting for a $N \times N$ block matrix that is uniformly partitioned with the GPU block size. Top-level parallelism is extracted from a main loop body depicted in Fig. 3. Approaching this algorithm in a bulk-synchronous manner, we find three supersteps in each iteration: (1) the LU factorization of the pivot block needs to be done by itself; (2) all TRSM operations can be scheduled independently; and (3) all GEMM operations can be scheduled independently. In each superstep, we observe that the multicore processor, being three times slower than the GPU, should be utilized for roughly one task in four. Any task that is assigned to the multicore processor is dynamically refined to its block size b, and these subdivided tasks are scheduled to the cores. Regarding the LU factorization of the pivot block, in our current setup, the GPU is incapable of this operation, so it is assigned to the multicore processor.

The scheduling of these three supersteps over the two devices is illustrated in Fig. 5. As stated, the first LU task is not processed on the GPU but redirected to the multicore processor. To use all cores, the block is subdivided and through a recursive call



(a) Multi-level partitions resulting from adaptive subdivision of tasks



(b) Heterogeneous task scheduling

Figure 5: Workloads are partitioned to heterogeneous devices based on their performance ratio. When a task is executed on a multicore processor, corresponding blocks are subdivided and create smaller tasks calling recursive block algorithms.

of a block LU factorization, fine grain tasks are created and scheduled to the multicore processor. None of the so created smaller tasks are scheduled to the GPU. The TRSM and GEMM operations are executed in separate bulk-synchronous supersteps. For instance, the subdivision of a single GEMM task ($C = A \times B$) using 2×2 block matrices creates 16 smaller GEMM tasks.

$$\left(\begin{array}{c|c} C^{(1,1)} & C^{(1,2)} \\ \hline C^{(2,1)} & C^{(2,2)} \end{array}\right) = \left(\begin{array}{c|c} A^{(1,1)} & A^{(1,2)} \\ \hline A^{(2,1)} & A^{(2,2)} \end{array}\right) \left(\begin{array}{c|c} B^{(1,1)} & B^{(1,2)} \\ \hline B^{(2,1)} & B^{(2,2)} \end{array}\right)$$

4 Tools

It has been long recognized that hand-coding of low level kernels is unlikely to be a match to the use of expertly developed library software, both in code quality and

resulting performance. With heterogeneous architectures where each component can have its own programming model this is even more true, and we limit our focus to developing the framework that lets these libraries work together harmoniously. Here, we describe the architecture-specific library tools on which we rely.

4.1 `libflame`

The `libflame` library [27] is a modern alternative to the widely used LAPACK library. It uses an Application Programming Interface (API) so that algorithms represented in code closely resemble the algorithms in Fig. 2. Using different algorithmic variants, the library can be easily tuned for various machine configurations. An extension of this API, Formal Linear Algebra Scalable Hierarchical API (FLASH) [11], allows the specification of and computation with hierarchically partitioned matrices (matrices of matrices). It is this API that facilitates the implementation of algorithm-by-blocks. A runtime system, SuperMatrix [6], is part of `libflame` and can be optionally used to schedule algorithm-by-blocks to multicore and/or multiGPU architectures. This runtime uses an explicitly formed DAG. There also exist other DLA libraries that offer similar functionality and performance to `libflame`, but these differ in that they do not stress programmability to the degree that `libflame` does. We believe, as do the developers of `libflame`, that while programmability has been considered as the second issue to performance, modern transient technologies require more productivity in porting libraries to new environments in a limited time, putting programmability center stage. Successful examples in this philosophy can be found many FLAME project related publications [13, 21].

4.2 Architecture specific BLAS or DLA libraries

Dense linear algebra libraries cast their computation in terms of operations supported by the BLAS interface, for portable high performance. The Nvidia CUBLAS [15] library is used when targeting GPUs with these operations, with a BLAS compatible interfaces. In our setup, tasks are most often BLAS operations with matrix blocks. When scheduling such an operation to a standard core, traditional BLAS are used. When targeting a GPU, the CUBLAS are used.

4.3 OpenMP

With cache coherence protocols that provide a hardware support to scalable parallelism on Symmetric Multi-Processings (SMPs), OpenMP has been widely used in many parallel applications. OpenMP has typically been used in loop-based work sharing since it was introduced in 1990s. Recently, OpenMP 3.0 in 2008 [17] supports task parallelism

to deal with increasing complexities in applications. In this work, OpenMP drives high-level task parallelism, where tasks are defined as BLAS or DLA functions executed on target devices. There are other parallel programming languages for runtime systems (*e.g*, Cilk [3], Intel TBB, and StarPU [2]) that offer advanced task scheduling, but OpenMP is almost unbeatable in terms of portability.

OpenMP 3.0 extended its programming model by adding explicit tasks. The new task scheme consists of two compiler directives: `#pragma omp task` to construct a new task and `#pragma omp taskwait` for the synchronization of invoked tasks. A task is defined with an executable code and its data environments. An important feature in OpenMP tasking is that a task can recursively create its children tasks. Tasks are scheduled in a Breadth First Search (BFS) manner; when a thread meets a directive `omp taskwait`, it suspends the executing task until its children tasks are completed. This feature makes it easy to employ OpenMP tasking to more irregular parallelism (e.g, recursion, pointer chasing, and tree traversal).

5 A Generic Task Scheduling Model for Heterogeneous Architectures

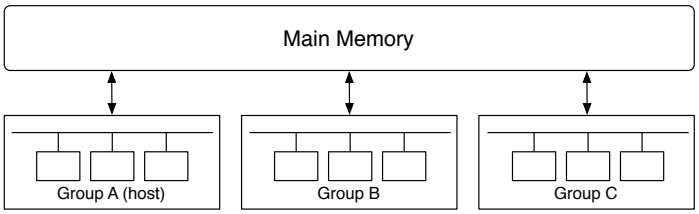
In this section, we give a formal description of the scheduling strategy that was presented in Section 3. In particular, we present a task scheduling model for general heterogeneous architectures that may integrate two or more different kinds of devices.

Our proposed scheduling strategy has a set-up phase and a use phase. The setup phase deals with hardware characteristics, and has to be performed only once for a given hardware configuration. The use phase then does the actual scheduling. In the setup phase, we characterize the heterogeneous compute node, such as the number of cores of the CPU and the number and type of attached devices. We also determine experimentally the optimal block sizes and the ratio of respective device performances. The scheduling phase then concerns itself with scheduling tasks according to capabilities and relative computing power of the various devices, and the dynamic subdivision of tasks.

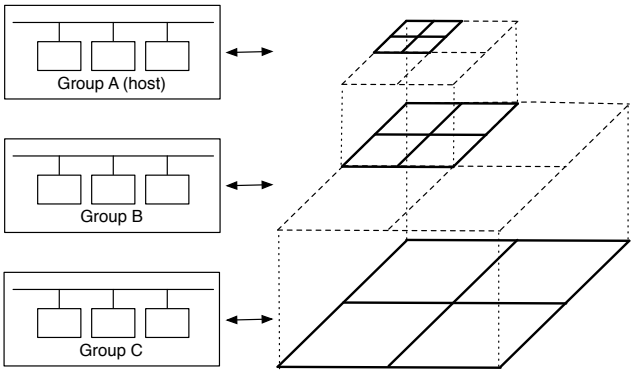
5.1 Node characterization

In this section, we describe the relevant parameters describing the heterogeneous node that are used in the scheduling algorithm.

Device structure. We target node-level heterogeneous architectures that integrate one or more general purpose multi-core processing units on shared main memory, with



(a) Network of devices



(b) Mapping device groups to multi-level hierarchical matrices

Figure 6: Relation of devices and hierarchical matrices

specialized co-processors or external accelerators grouped according to their blocksize and peak performance.

Accelerator units are typically equipped with fast local memory. In our computing model, all data are initially located in main memory. Necessary input blocks are sent to target devices, and corresponding output blocks in main memory are locked while tasks are being processed. As soon as a task has completed on the target device, the output blocks are written back to main memory and unlocked. For now, all data transfers are made through main memory, and direct peer-to-peer communication among devices is not allowed.

Fig. 6 describes a diagram that describes the hierarchy of devices, data, and tasks. Device groups are ordered with respect to their block sizes and mapped to different partition level of blocked matrices. Starting from initial blocks, recursive subdivision of blocks creates finer tasks that are distributed to the next group of devices. We also need to take into account the fact that devices may have different capabilities; for instance, in Section 3 the GPU is only capable of BLAS functions, so the scalar LU

factorization is performed on the CPU.

Block size and balance ratio. Next, we consider two parameters that are empirically determined: the block size and the balance ratio.

In our heterogeneous approach to scheduling, the granularity of tasks is dynamically adapted to executing devices through subdivision of blocks.

Let us consider the concerns:

- We want to have each device execute as efficiently as possible. For this reason, we want to pick the block size for a target device to be at least large enough so that tasks achieve near-asymptotic performance.
- We want to maximize load balance. For this reason, we should pick the block size no larger than necessary, since it is generally the case that the larger the block size, the worse the load balance.
- We will create teams of slower devices that together attain performance similar to the next faster device. For this reason, we may want to increase the block size for the faster device to ensure that all slower devices are kept busy with another task.

For $0 \leq k < \#levels$, we let P_k and N_k equal

P_k = Effective peak performance of a single device at level k
N_k = # of devices in a team at level k

then

$$\text{Balance ratio} = \beta_k = 1 + \frac{P_{\text{k}} \cdot N_{\text{k}}}{P_{\text{k+1}} \cdot N_{\text{k+1}}}$$

where level $k = 0$ corresponds to the fastest devices. This balance ratio β_k is used as follows: in the global work queue, tasks are assigned to the fastest device, but one in every β_0 is assigned to the slower device(s). This rule is applied recursively in case there are more than two device classes.

Although the above parameters are rough estimates, our experiments shows that they effectively control the global workload balance on heterogeneous devices.

5.2 Bulk-synchronous scheduling

We use a bulk synchronous approach to task scheduling. We assume that for a particular superstep the algorithm gives us a set of independent tasks. In our implementation, a parallel code is transparently derived from its algorithm using FLAME environments

```
def hier.LU_nopiv(FLA_Obj A) {
  FLA_Part_2x2(A, ATL, ATR,
                  ABL, ABR, From_Top_Left);

  while (ATL.length < A.length):
    Repart_2x2_to_3x3(ATL, ATR,   A00, A01, A02,
                                  A10, A11, A12,
                      ABL, ABR,   A20, A21, A22,
                      From_Bottom_Right);

    internal.LU_nopiv(A11(0,0))

    for j in A12.width:
      #pragma omp task firstprivate(A11, A12, j)
      internal.trsm(A11(0,0), A12(0,j))

    for i in A21.length:
      #pragma omp task firstprivate(A11, A21, i)
      internal.trsm(A11(0,0), A21(i,0))

    #pragma omp taskwait

    for j in A12.width:
      for i in A21.length:
        #pragma omp task
        internal.gemm(A21(i,0), A12(0,j), A22(i,j))

    #pragma omp taskwait

    Cont_with_3x3_to_2x2(ATL, ATR,  A00, A01, A02,
                                    A10, A11, A12,
                         ABL, ABR,  A20, A21, A22,
                         From_Top_Left)
```

Figure 7: A pseudo code for LU factorization with OpenMP tasking.

and OpenMP directives [28]. For example, Code 7 describes a pseudo code corresponding to the algorithm depicted in Fig. 2. This code creates tasks that are synchronized in supersteps, as shown in Section 3. Inside each superstep, task groups can then be dynamically subdivided and reinserted to accomodate slower devices.

5.3 Multi-level task scheduling

The objective of heterogeneous scheduling is to keep a proper workload balance across heterogeneous devices. This is done through recursive task inheritance: in each task group a portion of tasks determined by the performance ratio between two adjacent device groups is inherited to the next-level device group. Fig. 8 and Fig. 5 illustrate this procedure to recursively distribute tasks to different devices. Initially, a set of coarse grain tasks is available, delimited by superstep barriers. Tasks are locally executed or

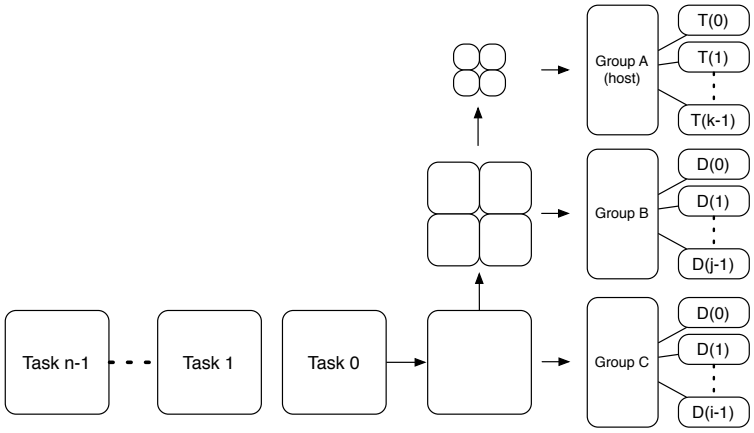


Figure 8: Tasks are initially created based on uniform blocks. After all devices in Group C are busy for processing given tasks, next task is subdivided and redistributed to devices in Group B. This process is recursively repeated until tasks reach the host device group.

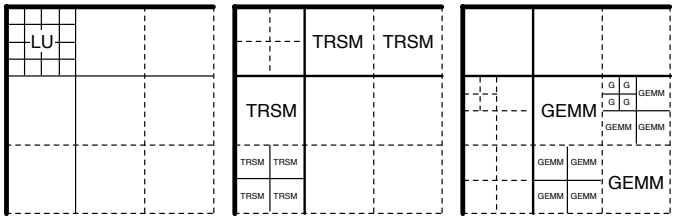


Figure 9: A scenario that blocks are recursively partitioned with three different block sizes during the first iteration of block LU factorization. Dot lines represent partitions on read-only blocks while solid lines partition output blocks that are exclusively managed in order to keep consistency in distributed data.

inherited to the next devices by performance ratios of device groups. Tasks are also inherited to the next devices if the encountered devices do not support relevant functions. We assume that the last device group is a general purpose multi-core processor with a full set of DLA libraries. For example, Fig. 9 illustrate a scenario for this recursive process among three different device groups. In this example, block LU tasks on diagonal blocks are always executed by a multi-core processor as we limit the use of other devices on BLAS functions only (*e.g*, supported by CUBLAS).

```
def op.apply(<list of blocks>, device):
  op.counter = 0
  for block in <list of blocks>:
    op.execute(block, device)

def op.execute(block, device):
  if op.counter % device.performance_ratio_to_next_device == 0:
    <list of blocks> =
    block.subdivide(get_next_block_size(block.size))
    op.apply(<list of blocks>, device.next_device)
  else:
    r_val = device.execute(op, block)
      if r_val == NOT_SUPPORT:
        <list of blocks> =
        block.subdivide(get_next_block_size(block.size))
        op.apply(<list of blocks>, device.next_device)
```

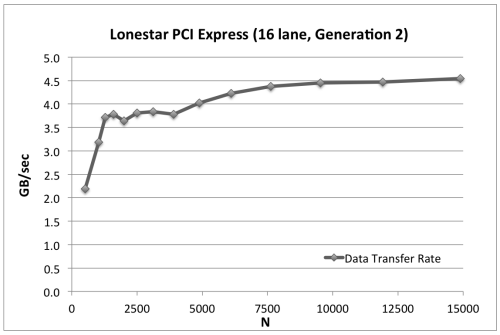
Figure 10: A pseudo code for multi-level task scheduling.

5.4 Data caching

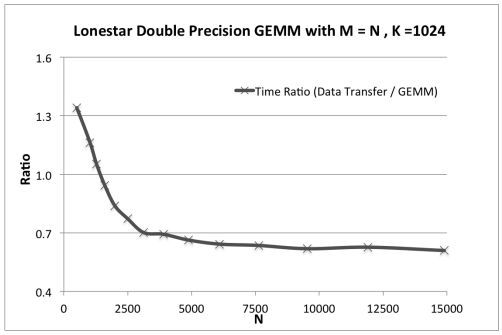
Heterogeneous task scheduling that relies on block algorithms inherently requires a large number of data transfers among devices. Such data transfers constitute overhead. In our model, data reside with a (shared memory) host which maintains consistency as blocks may be copied to local device memories. Fig. 11 illustrates data transfer overheads between main memory and a GPU, connected via a PCI Express bus. The graph shows there is a significant loss in GPU performance due to the limited bandwidth of a PCI Express bus. Such overheads still accounts for a certain portion in total numeric costs even for large problems; they reduce DGEMM performance by least 40%. This implies that a GPU can perform twice faster if all input blocks are already placed in its local memory as shown in Fig. 12. Hence, it is essential to reduce the number of communications between devices in order to achieve higher performance.

We implemented a simple mechanism, a software cache, to detect when data are still cached on a device so that it can be reused. Consistency of stored data in a local cache is maintained through the *write-through* policy (data are synchronously written in both local cache and main memory). This means that, after a task is processed on a device, any relevant updates on output blocks are immediately copied back to main memory. Communications between devices are governed by a host multi-core processor. Each device cache maintains its local copies consistent to data on main memory and there is no peer-to-peer communication between devices. This is described in Fig. 14.

Fig. 14 compares GPU performance with various data transfer regimes between main memory and the GPU. In this comparison, we assume that all matrices are initially created at main memory. The highest performance is achieved by a CUBLAS call,



(a) 16-lane PCI Express 2.0



(b) (Time for data transfer / Time for DGEMM)

Figure 11: Left: data transfer rate of PCI Express 2.0 as a function of matrix size. Right: cost ratio of data transfers over DGEMM depicted in Fig. 12.

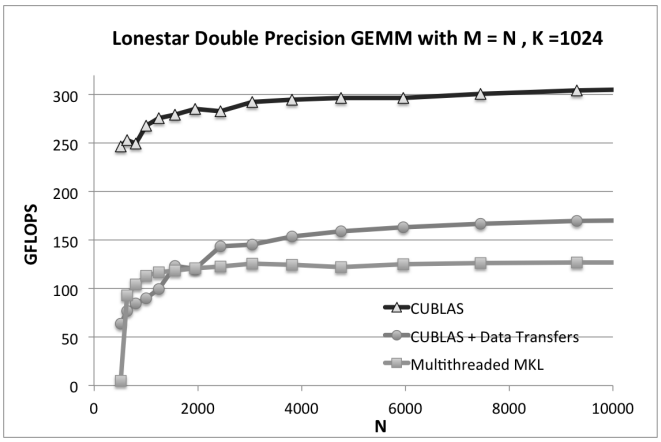


Figure 12: DGEMM performance with M = N and K = 1024 for a 12x Intel Xeon X5680 @ 3.33GHz processor and a single Fermi GPU M2070 at TACC Lonestar.

which takes only a single data transfer. Still, we note that this transfer results in halving of its peak performance (to 300 GFLOPS). We compare this to the performance of our scheduler, with and without cache. Clearly, even our simple caching strategy makes a noticeable difference, although we do not reach CUBLAS performance. However, our scheduler is fully general in what operations it can accommodate.

```
def device.execute(op, block):
  local_block_ptr = device.cache.lookup(block)
  if local_block_ptr == NULL:
    local_block_ptr = device.cache.copy_in(block)

  # invoke vendor provided libraries to execute a task
  r_val = execute(op, local_block_ptr)

  device.cache.copy_out(block)
  return r_val
```

Figure 13: A pseudo code for data management.

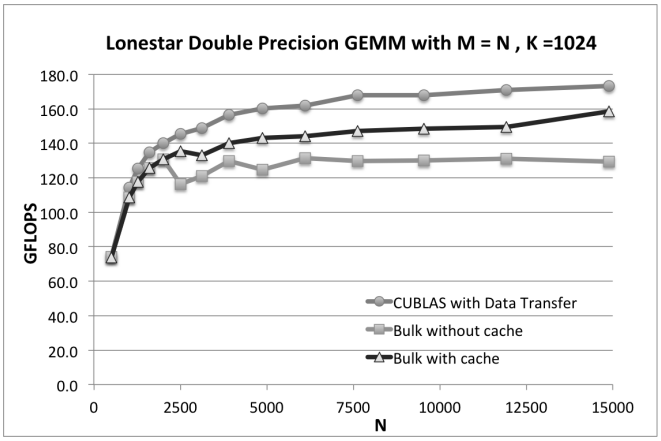


Figure 14: Performance of block DGEMM against CUBLAS DGEMM. Note that CUBLAS DGEMM includes communication overheads (Nvidia Fermi M2070 connected via PCI Express 2.0). Block DGEMM uses hierarchical matrices that are partitioned with a blocksize(2048) and a software cache(160 blocks).

6 Experimental Results

In this section, we evaluate our proposed block synchronous approach on three different platforms, for which details are tabulated in Fig. 15. Our hybrid DLA package, implementing LU on top of Bulk Synchronous Block Scheduling (BS2), is compared against other DLA libraries: MAGMA version 1.2 for a single GPU and Supermatrix for multiGPUs[2]. We summarize these libraries in Fig. 16; the most salient point is their treatment of devices. First of all we note that all three packages execute the point LU

2. The project that produced the MAGMA software also has reported on a version of the Quark scheduler that handles the heterogeneous case [12]. However, the software is in an experimental state and we were not able to use it.

	Longhorn	Lonestar	Lorca
Processors (Intel)	8x Core, Nehalem 2.53 GHz	2x Hex-Core, Westmere 3.33 GHz	2x Quad-Core, Harpertown 2.83 GHz
Memory	48 Gbytes	24 Gbytes	16 Gbytes
BLAS	Intel MKL 10.3		
DLA library	`libflame` ver 6192		
DGEMM Peak	10 GFLOPS/core	13.3 GFLOPS/core	11 GFLOPS/core
Interconnection BUS	PCI Express 2.0		
GPUs	2x NVIDIA Quadro FX 5800 (4 GB RAM)	2x NVIDIA Tesla M2070 (6 GB RAM)	4x NVIDIA Tesla S1070 (3 GB RAM)
Compiler	GNU 4.4.1	GNU 4.4.5	GNU 4.4.1
DGEMM peak/device	70 GFLOPS	300 GFLOPS	300 GFLOPS

Figure 15: Specification of testing machines. Longhorn and Lonestar are installed at TACC. Lorca is installed at the Universidad Jaume I De Castellón, Spain.

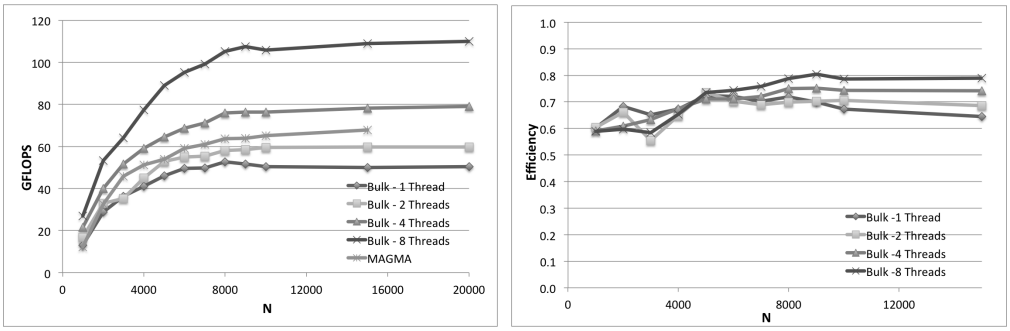
	MAGMA	SuperMatrix	Bulk (ours)
Algorithm	Blocked Alg.	Alg. by blocks	
Blocking	Auto-tune (64-256)	Single blk. (768)	Multi-level (GPUs 2048, Cores 256)
Data hosting	Device (6GB)	Main memory (24GB)	
Block Factorization	Single threaded		Multithread
BLAS	on GPUs		on GPUs and Cores
Scheduling	Static	Dynamic (DAG)	Dynamic (Bulk)

Figure 16: Distinct features in different DLA libraries. Specific parameters are based on the Lonestar machine equipped with Fermi GPUs.

factorization on the CPU; the packages then differ in how they treat BLAS operations, which in principle can be executed on either device. The packages also differ in how they offload data to the GPU.

- BS2 schedules any operation to any device, using only availability and load balancing as criterium. Task data are moved to a device as the operation is executed.
- SuperMatrix divides devices by functionality: the point LU factorization is exclusively executed on the CPU and BLAS operations are exclusively assigned to the GPU(s).
- MAGMA uses a reverse model: the top level operation is executed as much as possible on the GPU, and for this the matrix is transferred in full. (Note that this limits the size of the problem that can be handled.) Since the point LU factorization needs to be done on the CPU, data are moved there from the device. Since this involves very little data, the cost of this is negligible.

All experiments were performed in double precision and results are presented in GFLOPS (10^9 Floating-point Operations Per Second). We also report an efficiency measure de-



(a) Performance (b) Efficiency

Figure 17: [Longhorn] LU nopiv accelerated by a single GPU, using block sizes of 192 and 768 for the multicore team and GPU, respectively, and a 128 block software cache.

fined by

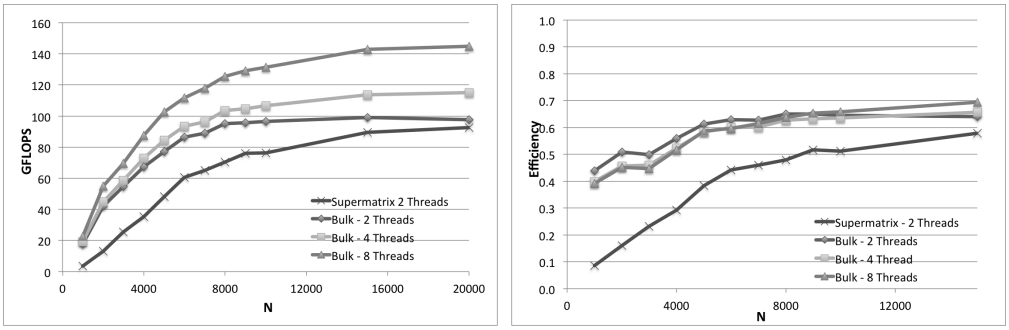
$$\text{efficiency} = \frac{\text{the system performance of hybrid algorithms}}{\text{sum of the performance of devices}},$$

which is appropriate for heterogeneous architectures: this formula measures the efficiency of integrating the components into one system, rather than comparing against some purely theoretical peak performance. For the GPU performance, we use that obtained from MAGMA.

6.1 Longhorn

Longhorn is a cluster where each node consists of an eight-core processor (Intel Nehalem) with two GPUs (Nvidia Quadro FX5800). Our computational model estimates the practical GPU performance as 60% of its on-device peak due to data transfer overheads in hybrid algorithms. The aggregate performance of the multicore processor exceeds that of a single GPU, so heterogeneous scheduling is set to distribute tasks almost evenly between the multicore processor and the GPUs, with the following parameters:

	Blocksize	β
2x GPU	768	2
8x CPU	192	-



(a) Performance (b) Efficiency

Figure 18: [Longhorn] LU nopiv accelerated by two GPUs, using block sizes of 192 and 768 for the multicore team and GPU, respectively, and a 128 block software cache.

A single GPU. Fig. 17 shows the absolute performance and relative efficiency (as defined above) of the proposed hybrid algorithms when only one GPU is used together with the multicore processor.

Some observations:

- Hybrid algorithms make good use of threads in the multicore processor: overall performance increases with the number of threads, and efficiency is more or less independent of this number (for large enough problems).
- Thanks to the performance added from a multicore processor, hybrid algorithms that only use high-level features in existing DLA libraries outperform the optimized implementations in MAGMA. This higher performance is attained despite the cost of numerous data transfers incurred by the algorithm-by-blocks. Performance for MAGMA is not reported for bigger problems due to limited local memory for a GPU.

Two GPUs. Fig. 18 compares the performance of hybrid multi-level LU factorization against `libflame` + SuperMatrix[3]. Regarding the hybrid efficiency, we compute the reference performance for multiGPUs by linearly scaling the single GPU performance from the use of MAGMA.

Fig. 18b shows that Supermatrix is less efficient for smaller problems since LU tasks are computed by a single thread. A blocksize(768) in this experiment is set to take more benefits from GPUs rather than a host multicore processor. Indeed, the performance

3. MAGMA version 1.2 also supports multiple GPUs statically distributing the matrix.

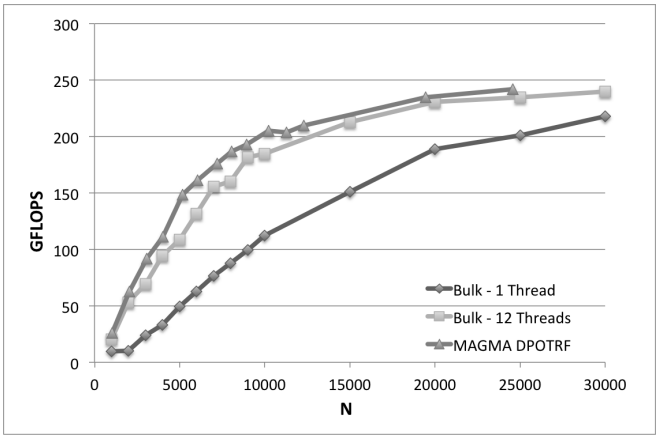


Figure 19: [Lonestar] Cholesky factorization accelerated by a single GPU, using block sizes of 256 and 2048 for the multicore team and GPU respectively, and a 160 block software cache.

on a multicore processor is traded to GPUs' performance until BLAS operations are dominant enough for large problems. However, multi-level scheduling dynamically refines tasks when they are executed on a multicore processor. This unique feature supplies fine grain tasks to a multicore processor while coarse grain tasks are still available for GPUs to keep their higher performance.

6.2 Lonestar

This machine is equipped with two *Fermi* GPUs (a more current Nvidia GPU), which delivers an order of magnitude higher double-precision performance (DGEMM 300 GFLOPS). A single GPU on this machine demonstrates much higher performance compared to the multicore team with 12 cores.

	Blocksize	β
2x GPU	2048	4
12x CPU	256	-

Due to such higher contrast in performance, hybrid scheduling is tuned for most tasks to flow into GPUs. Meanwhile, a host processor is responsible for the factorization of diagonal blocks and only some of the BLAS operations.

A single Fermi *GPU.* Fig. 19 reports performance for the Cholesky factorization executed by the proposed hybrid scheduling, compared to the MAGMA implementa-

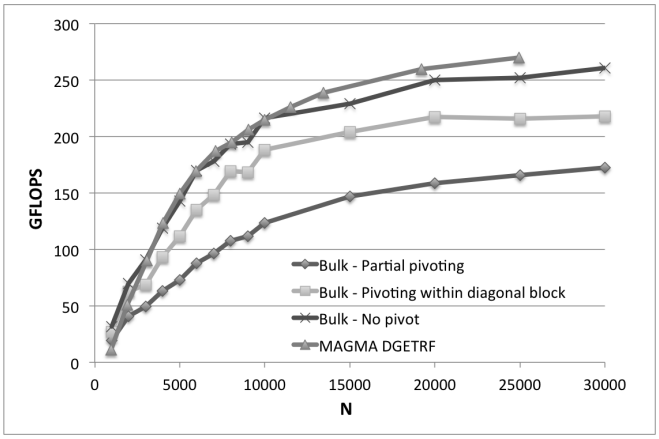


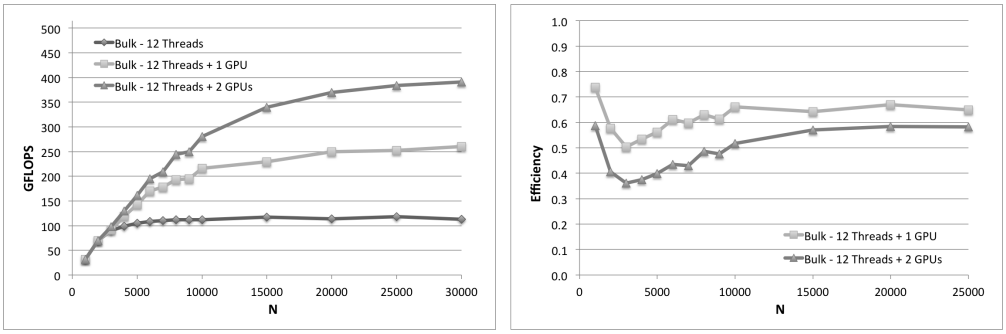
Figure 20: [Lonestar] LU factorization with various pivoting strategies with a single GPU acceleration, using block sizes of 256 and 2048 for the multicore team and GPU respectively, and a 160 block software cache.

tion. The hybrid performance on this machine is not so higher than the performance obtained by MAGMA from a single GPU because we did not implement Cholesky factorization on the GPUs but instead direct those tasks to the multicore processor. This lowers overall hybrid performance because Cholesky tasks on diagonal blocks are on the critical path, they are sequentially dispatched, and they create data traffic between the multicore team and the GPU.

Fig. 20 reports the effects of different pivoting strategies on the proposed hybrid LU factorization. Applying pivots on a multicore processor invalidates some blocks stored in a GPU software cache, which then triggers more data transfers. Pivoting within the diagonal block selects a pivot from a diagonal block reduces the number of blocks to be flushed in a soft-cache. Meanwhile, partial pivoting that examines an entire column invalidates almost all blocks stored in a GPU soft-cache. Although numerical stability is traded to performance, pivoting within a large block size that is typically used for GPUs can be considered in most dense matrix problems.

Two Fermi *GPUs.* Fig. 21 reports performance for hybrid LU factorization without pivoting. Decreasing efficiency for small problems is artificial because GPUs contribute nothing until the problem size is larger than the GPU block size.

Lorca This machine has an eight-core processor accelerated by four *Fermi* GPUs. In our hybrid computational model, four threads are now used for controlling the GPUs



(a) Performance (b) Efficiency

Figure 21: [Lonestar] LU nopiv accelerated by multiGPUs, using block sizes of 256 and 2048 for the multicore team and GPU respectively, and a 160 block software cache.

and the other four threads can execute tasks. However, performance obtained from the rest of four threads notably lags behind the much higher performance of the GPUs.

	Blocksize	β
4x GPU	2048	-
8x CPU	256	-

Dispatching tasks to them merely incurs more data transfers because blocks updated by a multicore processor invalidates device local copies. Considering the limited bandwidth of PCI Express bus that are shared by four GPUs, data transfer overheads are more dominant than for the other testing machines. Hence, a multicore processor is set to devote all its resources to factorize diagonal blocks only; no BLAS operation is allowed on a multicore processor. Fig. 22 reports how the hybrid scheduling algorithms cooperate with a eight-core machine accelerated by four Fermi GPUs.

7 Conclusion

We have presented a generic framework for scheduling on heterogeneous multicore architectures, and applied this to computing with dense matrices. The proposed hybrid scheduling is based on multi-level task parallelism where tasks are dynamically identified and adapted to target devices. The main goal of hybrid multi-level schemes is to run each device with its highest runtime efficiency so as to boost overall system performance. Workloads are dynamically balanced while distributing non-uniform block tasks to heterogeneous devices. The proposed hybrid scheduling is flexible and can be

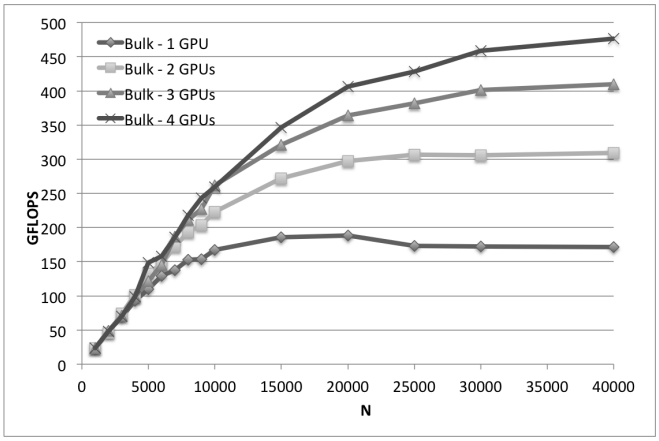


Figure 22: [Lorca] LU nopiv accelerated by multiGPUs, using block sizes of 256 and 2048 for the multicore team and GPU respectively, and a 80 block software cache.

extended to work with diverse heterogeneity. The multi-level scheduling is also tunable with a few parameters characterizing heterogeneous architectures.

By way of demonstration of our scheduling strategy, we implemented LU factorization by using high-level device specific DLA library components written by experts. Experimental results shows that hybrid scheduling can achieve 70% of each device's peak performance. Testing the algorithms on different testbeds, our general approach was seen to perform better or equivalent to MAGMA and other DLA libraries.

Our scheduling approach can accomodate current as well as future architecture designs. For example, one could consider adding Out-Of-Core (OOC) devices to a hybrid GPU model. Using random access devices such as Solid State Devices (SSDs) and fast peripherals, the hybridization with OOC devices are more attractive. In our frameworks, OOC device is considered as a processing device although it does not compute anything but upload data from OOC to in-core memory. Using multi-level hierarchical matrices, larger blocks are efficiently pre-loaded. This can significantly reduce the number of access to OOC devices.

Acknowledgement

We thank TACC at the University of Texas at Austin, and the Universidad Jaume I De Castellón for allowing to use their equipments in this work. This research was sponsored by National Science Foundation (NSF) under grant no. 0904907. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

The code that this paper describes has been developed based on `libflame` and OpenMP. Codes are available under the GNU Lesser General Public License (LGPL) for the non-commercial use at `http://code.google.com/p/uhm`.

References

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, J. Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):1–5, 2009.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and P.A. Wacrenier. STARPU : A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 37, pages 207–216, August 1995.

[4] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009.

[5] Ernie Chan, Robert van de Geijn, and Andrew Chapman. Managing the complexity of lookahead for LU factorization with pivoting. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures - SPAA '10*, pages 200–208, New York, New York, USA, 2010. ACM Press.

[6] Ernie Chan, Field G. van Zee, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Satisfying your dependencies with Supermatrix. In *2007 IEEE International Conference on Cluster Computing*, pages 91–99. IEEE, 2007.

[7] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, 1988.

[9] Fred G. Gustavson, Isak Jonsson, Bo Kagström, and Per Ling. Towards peak performance on hierarchical SMP memory architectures - new recursive blocked data formats and BLAS. In *Parallel Processing for Scientific Computing*, 1999.

[10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, 1979.

[11] Tze Meng Low. An API for manipulating matrices stored by blocks. Technical report, FLAME Working Note 12, TR-2004-15, The University of Texas at Austin, Depar, 2004.

[12] Hatem Ltaief, Stanimire Tomov, Rajib Nath, and Jack Dongarra. Hybrid multicore Cholesky factorization with multiple GPU accelerators. *IEEE Transaction on Parallel and Distributed Systems*, 2010.

[13] Bryan Marker, Ernie Chan, Jack Poulson, R. van de Geijn, R.F. Van der Wijngaart, T.G. Mattson, and T.E. Kubaska. Programming many-core architectures - a case study: dense matrix computations on the Intel SCC processor. *Concurrency and Computation: Practice and Experience*, 2011.

[14] R. Nath, S. Tomov, and J. Dongarra. An Improved MAGMA GEMM For Fermi Graphics Processing Units. *International Journal of High Performance Computing Applications*, 24(4):511–515, November 2010.

[15] Nvidia. *CUBLAS User Guide*. `http://developer.nvidia.com`.

[16] Nvidia. *CUDA C Programming Guide*. `http://developer.nvidia.com`.

[17] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. `http://www.openmp.org`, 2008.

[18] Enrique Quintana-Ortí, Gregorio Quintana-Ortí, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[19] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, volume 44, pages 121–130, February 2009.

[20] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level Parallelism. *ACM Transactions on Mathematical Software*, 36(3):1–26, July 2009.

[21] Gregorio Quintana-Ortí and Robert van de Geijn. Level-3 BLAS on a GPU: Picking the Low Hanging Fruit. Technical report, FLAME Working Note 37, DICC 2009-04-01, The University of Texas at Austin, Department of Computer Sciences, 2009.

[22] Fengguang Song, S. Tomov, and Jack Dongarra. Efficient support for matrix computations on heterogeneous multi-core and multi-GPU architectures. Technical report, LAPACK Working Note 250, UT-CS-11-669, The University of Tennessee at Knoxville, EECS department, 2011.

[23] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, and Yungang Bao. Fast implementation of DGEMM on Fermi GPU. In *SC2011- International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 35:1–11. ACM, 2011.

[24] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[25] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–839, August 2002.

[26] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www. lulu. com`, 2008.

[27] Field G. van Zee. *libflame: The Complete Reference*. `http://www.lulu.com`, 2009.

[28] Field G. van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable parallelization of FLAME code via the workqueuing model. *ACM Trans. Math Software*, 34(2):10:1–29, 2008.

[29] Vasily Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 31:1–11, 2008.