Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs

Emmanuel Agullo[*] Cédric Augonnet[*] Jack Dongarra[†]
Hatem Ltaief[†] Raymond Namyst[*] Samuel Thibault[*]
Stanimire Tomov[†‡ §]

13th September, 2010

In this chapter, we present a *hybridization methodology* for the development of linear algebra software for GPUs. The methodology is successfully used in MAGMA – a new generation of linear algebra libraries, similar in functionality to LAPACK, but extended for hybrid, GPU-based systems. Algorithms of interest are split into computational tasks. The tasks' execution is scheduled over the computational components of a hybrid system of multicore CPUs with GPU accelerators using StarPU – a runtime system for accelerator-based multicore architectures. StarPU enables to express parallelism through sequential-like code and schedules the different tasks over the hybrid processing units. The productivity becomes then **fast** and **cheap** as the development is high level, using existing software infrastructure. Moreover, the resulting hybrid algorithms are **better** performance-wise than corresponding homogeneous algorithms designed exclusively for either GPUs or homogeneous multicore CPUs.

[*]INRIA, LaBRI, University of Bordeaux, France
[†]Innovative Computing Laboratory, University of Tennessee, Knoxville TN 37996
[‡]Authors are listed in alphabetical order.
[§]Preprint submitted to GPU-GEMs.

1 Introduction, Problem Statement, and Context

The large scale enabling of GPU-based architectures for high performance computational science depends on the successful development of fundamental numerical libraries. Major issues in terms of developing new algorithms, programmability, reliability, and user productivity must be addressed on these systems. At the same time, it becomes paramount to efficiently schedule algorithms over heterogeneous platforms to take advantage to the fullest extent from the computational power of their hybrid components. This chapter describes a methodology on how to develop these algorithms and libraries in the area of linear algebra (LA). The impact of developing and making LA libraries available is far reaching because many science and engineering applications depend on them; these applications will not perform well unless the linear algebra libraries perform well.

The hybridization methodology is twofold. First, we design highly efficient algorithms optimized to run on a single GPU and its CPU host. This approach has been successfully used in the Matrix Algebra on GPU and Multicore Architectures (MAGMA) project [1] and the libraries stemming from it. MAGMA is designed to be similar to the popular LAPACK library in functionality, data storage and interface, to allow scientists to effortlessly port any LAPACK-relying software components to take advantage of new hybrid architectures. Second, we use the hybrid algorithms and kernels from the first approach as building blocks of higher-level algorithms designed for hybrid systems of multicore CPUs with multi-GPU accelerators. We use the StarPU [2] runtime system to schedule those tasks on the computational units. StarPU enables to express parallelism through sequential-like code and schedules the different tasks over the hybrid processing units. We illustrate this approach with the Cholesky factorization, a fundamental and representative LA algorithm.

2 Core Method

Our goal is to run numerical algorithms as fast as possible on complex architectures composed of multicore CPUs with GPU accelerators in a portable way. The proposed method is in three steps. The first step consists of writing the numerical algorithm at a high-level of abstraction as a sequence of multiple tasks of fine granularity; a task can be executed on a CPU core, on a GPU or on both resources simultaneously (hybrid task). The second step consists of providing high performance kernels with two interfaces, CPU and GPU, implementing each task. These kernels may already be available (such as vendor kernels) or may need to be designed (when highly optimized vendor kernels are not available). The final step requires to integrate the high-level algorithm along with the kernel in a runtime system. The runtime system is then in charge of scheduling the different tasks onto the processing units without violating the dependences of the high-level algorithm while ensuring data availability and coherency.

We illustrate our method with the Cholesky factorization of dense matrices. This algorithm can be decomposed into fine granularity BLAS calls. We use CPU BLAS kernels from Intel MKL [3] and GPU BLAS kernels from the MAGMA library. We schedule the operations using the StarPU runtime system that exploits all computational CPU and GPU resources in a portable way on complex heterogeneous machines, hiding the low-level complexity.

3 Algorithms, Implementations, and Evaluations

3.1 Cholesky Factorization

The Cholesky factorization (or Cholesky decomposition) of an $n \times n$ real symmetric positive definite matrix A has the form $A = LL^T$, where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. This factorization is mainly used as a first step for the numerical solution of linear equations $Ax = b$, where A is a symmetric, positive definite matrix. Such systems arise often in physics applications, where A is positive definite due to the nature of the modeled physical phenomenon. The reference implementation of the Cholesky factorization for machines with hierarchical levels of memory is part of the LAPACK library. It consists of a succession of panel (or block column) factorizations followed by updates of the trailing submatrix. The algorithm can easily be parallelized using a fork-join approach since each update – consisting of a matrix-matrix multiplication – can be performed in parallel (fork) but that a synchronization is needed before performing the next panel factorization (join). A variant of this algorithm is well suited for being executed onto a single GPU. We will present it in Section 3.2. In the multi-GPU case, the number of synchronizations of this algorithm would be a prohibitive bottleneck for performance. Instead, the panel factorization and the update of the trailing submatrix are broken

```
for (k = 0;  k < Nt; k++)
  A[k][k] = potrf(A[k][k])
  for (m = k+1;  m < Nt; m++)
    A[m][k] = trsm(A[k][k],A[m][k])
  for (m = k+1;  m < Nt; m++)
    for (n = k+1;  n < m; n++)
      A[m][n] = gemm(A[m][k],A[n][k],A[m][n])
    A[m][m] = syrk(A[m][k],A[m][m])
```

Figure 1: Tile Cholesky decomposition of a matrix A composed of Nt $\times$ Nt tiles.

into smaller tasks that operate on square submatrices of fine granularity, so called *tiles*. The corresponding algorithm, initially developed for multicore architectures [4, 5], is called *tile Cholesky factorization*. It consists of the three nested loops of Figure 1 relying on four BLAS and LAPACK kernels. Since the single-GPU hybrid algorithms we present can be used as kernels for the tasks of multi-GPU algorithms, hybridization is twofold: at the task level and between tasks.

3.2 Kernels for single CPU cores and single GPUs

Once the algorithm has been split into smaller tasks, high performance CPU or GPU kernels implementing each task need to be provided. If both the CPU and GPU implementations of a kernel are provided, the runtime will furthermore have the opportunity to schedule the task on either hardware. These kernels may already be available (such as vendor kernels) or may need to be designed (when highly optimized vendor kernels are not available). In the case of the tile Cholesky factorization, the four kernels needed are the LAPACK and BLAS routines: *spotrf*, *sgemm*, *strsm* and *ssyrk*. Since the corresponding highly tuned implementations are already provided by the CPU vendor within the Intel MKL 10.1 library, we directly use the corresponding sequential routines from that library. For GPUs, NVIDIA provides high performance implementations of the *sgemm* and *strsm* and *ssyrk* routines within the CUBLAS library. In different cases, depending on hardware and problem sizes, we use these kernels either from CUBLAS or MAGMA BLAS. The LAPACK *spotrf* routine is not provided by NVIDIA. We use the highly tuned version that we developed and made freely available in the MAGMA [1] library. In the rest of this section, we present the underlying algorithm as roadmap to design high performance hybrid algorithms for a single CPU core enhanced by a GPU.

```
 1 for (j = 0;  j < *n; j += nb) {
 2   jb = min(nb, *n-j);
 3   cublasSsyrk('l','n', jb, j, -1, da(j,0),*lda, 1, da(j,j),*lda);
 4   cudaMemcpy2DAsync(work, jb*sizeof(float), da(j,j), *lda*sizeof(float),
 5                 sizeof(float)*jb, jb, cudaMemcpyDeviceToHost, stream[1]);
 6   if (j + jb < *n)
 7      cublasSgemm('n','t', *n-j-jb, jb, j, -1, da(j+jb,0), *lda, da(j,0),
 8                  *lda, 1, da(j+jb,j), *lda);
 9   cudaStreamSynchronize(stream[1]);
10   spotrf_("Lower", &jb, work, &jb, info);
11   if (*info != 0)
12      *info = *info + j, break;
13   cudaMemcpy2DAsync(da(j,j), *lda*sizeof(float), work, jb*sizeof(float),
14                 sizeof(float)*jb, jb, cudaMemcpyHostToDevice, stream[0]);
15   if (j + jb < *n)
16      cublasStrsm('r','l','t','n', *n-j-jb, jb, 1, da(j,j), *lda,
17                  da(j+jb,j), *lda);
18 }
```

Figure 2: Hybrid Cholesky factorization for single CPU-GPU pair (*spotrf*).

Hybrid Cholesky Factorization for a single GPU Figure 2 gives the hybrid Cholesky factorization implementation for a single GPU. Here `da` points to the input matrix that is on the GPU memory, `work` is a work-space array on the CPU memory, and `nb` is the blocking size. This algorithm assumes the input matrix is stored in the leading n-by-n lower triangular part of `da`, which is overwritten on exit by the result. The rest of the matrix is not referenced. Compared to the LAPACK reference algorithm, the only difference is that the hybrid one has three extra lines – 4, 9, and 13. These extra lines implement our intent in the hybrid code to have the jb-by-jb diagonal block starting at da(j,j) factored on the CPU, instead of on the GPU. Therefore, at line 4 we send the block to the CPU, at line 9 we synchronize to insure that the data has arrived, factor it next on the CPU using call to LAPACK at line 10, and send the result back to the GPU at line 13. Note that the computation at line 7 is independent of the factorization of the diagonal block, allowing us to do these two tasks in parallel on the CPU and on the GPU. This is implemented by "scheduling" first the *sgemm* (line 7) on the GPU; this is an asynchronous call, hence the CPU continues immediately with the *spotrf* (line 10) while the GPU is running the *sgemm*.

To summarize, the following is achieved with this algorithm:

- The LAPACK Cholesky factorization is split into tasks;
- Large, highly data parallel tasks, suitable for efficient GPU computing, are statically "scheduled" for execution on the GPU;
- Small, inherently sequential *spotrf* tasks (line 10), not suitable for efficient GPU computing, are executed on the CPU using LAPACK;

- Small CPU tasks (line 10) are overlapped by large GPU tasks (line 7);
- Communications are asynchronous to overlap them with computation;
- Communications are in a surface-to-volume ratio with computations – sending nb^2 elements at iteration j is tied to O$(nb \times j^2)$ flops, $j \geq nb$.

3.3 Running on multiple GPUs using StarPU

Accelerator-based platforms, such as multicore architectures enhanced by GPU accelerators, are complex to program. Not only an algorithm needs to be split into smaller tasks to enable the concurrent use of all the computational units, but the data coherency must be ensured between the memories of the different units. This complexity may be delegated to a runtime system, StarPU. This methodology allows the programmer to focus on *what* to do (e.g., choosing a scheduling strategy) while the runtime system takes care of *how* to do it efficiently (e.g., ensuring data transfers and coherency). The monitoring of the system by the runtime furthermore allows the design of efficient, adaptative strategies. Because low-level technical issues are not the preoccupation of the programmer anymore, the productivity is very high.

Once the algorithm has been conceptually split into tasks, the program may be written as a succession of task insertions. A task being a function working on a data, those two notions are central to StarPU. First, all data are first registered into StarPU. Once a data is registered, the application does not access it anymore through its memory address but through a StarPU abstraction, the *handle*. The handle does not change during the whole execution. If the runtime decides to schedule a task onto a unit that does not have a valid copy of a data, it will take care of the data movement. The pointer to the data will be internally updated but the handle exposed to the application will remain unchanged. StarPU transparently guarantees that when a task needs to access a piece of data, it will be given a pointer to a valid data replicate. Second, assuming that an implementation of the computational kernels is provided for each device (CPU core and GPU), a multi-version kernel, the *codelet*, is defined on top of them. In the end, a task can be defined independently of the device as a codelet working on handles. The tasks are then executed according to a scheduling strategy that can be either selected among pre-existing policies, or specifically designed for the task. We now show in details how to program the tile Cholesky factorization on top of StarPU.

```
float *tile[mt][nt];                          // Actual memory pointers
starpu_data_handle tile_handle[mt][nt]; // StarPU abstraction

for (n = 0; n < nt; n++) //loop on cols
    for (m = 0; m < mt; m++) //loop on rows
        starpu_matrix_data_register(&tile_handle[m][n], 0,
                                    &tile[m][n], M, M, N, sizeof(float));
```

Figure 3: Registration of the tiles as handles of matrix data type.

Initialization. When initializing StarPU with `starpu_init`, StarPU automatically detects the topology of the machine and launches one thread per processing unit to execute the tasks.

Data registration. Each tile is registered into StarPU to be associated to a handle. As shown in Figure 3, the `tile_handle[m][n]` StarPU abstraction is obtained from each actual memory pointer, `tile[m][n]`. Several data types are pre-defined for the handles. Here, tiles are registered as matrices since a submatrix is itself a matrix.

Codelets definition. As shown at lines 38-42 for the `sgemm_codelet` in Figure 4, a codelet is a structure that describes a multi-versioned kernel (*sgemm* here). It contains pointers to the functions that implement the kernel on the different types of units: lines 1-14 for the CPU and 16-30 for the GPU. The prototype of these functions is fixed: an array of pointers to the data interfaces that describe the local data replicates, followed by a pointer to some user-provided argument for the codelet. The `STARPU_MATRIX_GET_PTR` is a helper function that takes a data interface in the matrix format and returns the address of the local copy. Function `starpu_unpack_cl_args` is also a helper function that retrieves the arguments stacked in the `cl_arg` pointer by the application. Those arguments are passed when the tasks are inserted.

Tasks insertion. In StarPU, a task consists of a codelet working on a list of handles. The access mode (e.g., read-write) of each handle is also required so that the runtime can compute the dependences between tasks. A task may also take values as arguments (passed through pointers). A task is inserted with the `starpu_insert_Task` function. [1] Lines 33-41 in Figure 5 shows how the *sgemm* task is inserted. The first argument is the codelet, `sgemm_codelet`. The following arguments are either values (keyword VALUE) or handles (when an access mode is specified). For instance, a value is specified at line 34, corresponding to the content of the `notrans`

[1]Other interfaces not discussed here are available.

```
void sgemm_cpu_func(void *descr[], void *cl_arg) {
    int transA, transB, M, N, K, LDA, LDB, LDC;
    float alpha, beta, *A, *B, *C;

    A = STARPU_MATRIX_GET_PTR(descr[0]);
    B = STARPU_MATRIX_GET_PTR(descr[1]);
    C = STARPU_MATRIX_GET_PTR(descr[2]);

    starpu_unpack_cl_args(cl_arg, &transA, &transB, &M,
                          &N, &K, &alpha, &LDA, &LDB, &beta, &LDC);

    sgemm(CblasColMajor, transA, transB, M, N, K,
          alpha, A, LDA, B, LDB, beta, C, LDC);
}

void sgemm_cuda_func(void *descr[], void *cl_arg) {
    int transA, transB, M, N, K, LDA, LDB, LDC;
    float alpha, beta, *A, *B, *C;

    A = STARPU_MATRIX_GET_PTR(descr[0]);
    B = STARPU_MATRIX_GET_PTR(descr[1]);
    C = STARPU_MATRIX_GET_PTR(descr[2]);

    starpu_unpack_cl_args(cl_arg, &transA, &transB, &M,
                          &N, &K, &alpha, &LDA, &LDB, &beta, &LDC);

    cublasSgemm(magma_const[transA][0], magma_const[transB][0],
                M, N, K, alpha, A, LDA, B, LDB, beta, C, LDC);
    cudaThreadSynchronize();
}

struct starpu_perfmodel_t cl_sgemm_model = {
    .type   = STARPU_HISTORY_BASED,
    .symbol = "sgemm"
};

starpu_codelet sgemm_codelet = {
    .where     = STARPU_CPU|STARPU_CUDA, // who may execute?
    .cpu_func  = sgemm_cpu_func, // CPU implementation
    .cuda_func = sgemm_cuda_func, // CUDA implementation
    .nbuffers  = 3, // number of handles accessed by the task
    .model     = &cl_sgemm_model // performance model (optional)
};
```

Figure 4: A codelet implementing *sgemm* kernel.

```
void hybrid_cholesky(starpu_data_handle **Ahandles,
                     int M, int N, int Mt, int Nt, int Mb)
{
 int lower = Lower;     int upper = Upper; int right = Right;
 int notrans = NoTrans; int conjtrans = ConjTrans;
 int nonunit = NonUnit; float one = 1.0f; float mone = -1.0f;

 int k, m, n, temp;
 for (k = 0; k < Nt; k++)
 {
  temp = k == Mt-1 ? M-k*Mb : Mb ;
  starpu_Insert_Task(spotrf_codelet,
    VALUE, &lower, sizeof(int), VALUE, &temp, sizeof(int),
    INOUT, Ahandles[k][k],      VALUE, &Mb, sizeof(int), 0);

  for (m = k+1; m < Nt; m++)
  {
   temp = m == Mt-1 ? M-m*Mb : Mb ;
   starpu_Insert_Task(strsm_codelet,
     VALUE, &right, sizeof(int),     VALUE, &lower, sizeof(int),
     VALUE, &conjtrans,sizeof(int), VALUE, &nonunit, sizeof(int),
     VALUE, &temp, sizeof(int),      VALUE, &Mb, sizeof(int),
     VALUE, &one, sizeof(float),     INPUT, Ahandles[k][k],
     VALUE, &Mb, sizeof(int),        INOUT, Ahandles[m][k],
     VALUE, &Mb, sizeof(int),        0);
  }

  for (m = k+1; m < Nt; m++)
  {
   temp = m == Mt-1 ? M-m*Mb : Mb;
   for (n = k+1; n < m; n++)
   {
    starpu_Insert_Task(sgemm_codelet,
      VALUE, &notrans, sizeof(notrans),
      VALUE, &conjtrans, sizeof(conjtrans),
      VALUE, &temp, sizeof(int),    VALUE, &Mb, sizeof(int),
      VALUE, &Mb, sizeof(int),      VALUE, &mone, sizeof(float),
      INPUT, Ahandles[m][k],        VALUE, &Mb, sizeof(int),
      INPUT, Ahandles[n][k],        VALUE, &Mb, sizeof(int),
      VALUE, &one, sizeof(one),     INOUT, Ahandles[m][n],
      VALUE, &Mb, sizeof(int),      0);
   }

   starpu_Insert_Task(ssyrk_codelet,
     VALUE, &lower, sizeof(int),  VALUE, &notrans, sizeof(int),
     VALUE, &temp,  sizeof(int),  VALUE, &Mb, sizeof(int),
     VALUE, &mone, sizeof(float), INPUT, Ahandles[m][k],
     VALUE, &Mb, sizeof(int),     VALUE, &one, sizeof(float),
     INOUT, Ahandles[m][m],       VALUE, &Mb, sizeof(int), 0);
  }
 }

 starpu_task_wait_for_all();
}
```

Figure 5: Actual implementation of the tile Cholesky hybrid algorithm with StarPU

variable. On the right of line 40, the handle of the tile (m,n) is passed in read-write mode (key-word INOUT). Figure 5 is a complete implementation of the tile Cholesky algorithm from Figure 1, showing the ease of programmability.

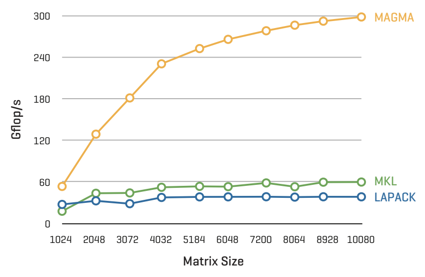
Finalization. Once all tasks have been submitted, the application can perform a barrier using the `starpu_task_wait_for_all()` function (line 53 in Figure 5). When it returns, we can stop maintaining data coherency and put the tiles back into main memory by unregistering the different data handles. Calling `starpu_shutdown()` releases all the resources.

Choice or design of a scheduling strategy. Once the above steps have been completed, the application is fully defined and can be executed as it is. However, the choice of a strategy may be critical for performance. StarPU provides several built-in pre-defined strategies that the user can select during the initialization, depending on the specificities and requirements of the application. When the performance of the kernels is stable enough to be predictable directly from the previous executions (as it is the case with Tile Cholesky factorization), one may associate an auto-tuned history-based performance model to a codelet as shown at lines 32-35 and 42 in factorization), one should associate an auto-tuned history-based performance model to a codelet as shown on lines 32-35 and 42 in Figure 4. If all codelets are associated to a performance model, it is then possible to schedule the tasks according to their expected termination time. The most efficient scheduling strategy (among those available in StarPU) for the Cholesky factorization is based on the standard Heterogeneous Earliest Finish Time (HEFT) [6] scheduling heuristic which aims at minimizing the termination time of the tasks on heterogeneous platforms. Given the impact of data transfers, especially when it comes to multiple accelerators, we extended this policy to take data transfer into account and keep it as low as possible. StarPU also provides a framework to develop *ad hoc* scheduling strategies in a high-level way, but the methodology to write a scheduler in StarPU is out of the scope of this description.

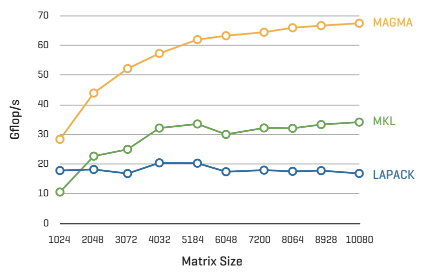
4 Final Evaluation

4.1 Performance results using single GPU

The performance of the hybrid Cholesky factorization from Figure 2 simultaneously running on a NVIDIA GTX 280 GPU and on one core of a dual socket quad-core Intel Xeon running at 2.33 GHz is given in Figure 6. The factorization runs asymptotically at 300 Gflop/s in single and almost 70 Gflop/s in double precision arithmetic. The performance has been evalu-



(a) Single Precision.



(b) Double Precision.

Figure 6: Parallel performance of MAGMA's hybrid Cholesky on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz

ated on a number of NVIDIA GPUs, including the newest Fermi C2050. For example, for matrix size of $9,984$ the hybrid algorithm runs at 631 GFlop/s on the GTX480 and 507 GFlop/s on the C2050. The double precision performance on the C2050 is 240 GFlop/s for the same matrix size.

4.2 Multi-GPU Overall Performance

We now present performance results for a hybrid system of eight Intel Nehalem X5550 CPU cores running at 2.67 GHz enhanced with three NVIDIA FX5800 GPUs running at 1.30 GHz. The overall performance of the method depends on the choice of the tile size which trades off parallelism and kernel performance. For the sake of simplicity, we chose a constant tile size. We empirically chose it equal to 960, which is well-suited for large matrices. The corresponding to this tile size matrix-matrix multiplication (*sgemm*) performances are respectively 20 Gflop/s per core (obtained with the Intel MKL 10.1 library) and 333 Gflop/s per GPU (obtained with the MAGMA 0.2 library). A GPU kernel needs a dedicated CPU core to be executed. Therefore, the node can be viewed as three GPU/CPU pairs and five supplementary available CPUs. If the supplementary CPUs are not used, a performance upper bound of the node is equal to 1000 Gflop/s; if they are used, the upper bound becomes equal to 1100 Gflop/s.

Using the three GPUs the Cholesky factorization achieves 780 Gflop/s (Figure 7). This corresponds to a perfect speedup equal to 3. The use of the five

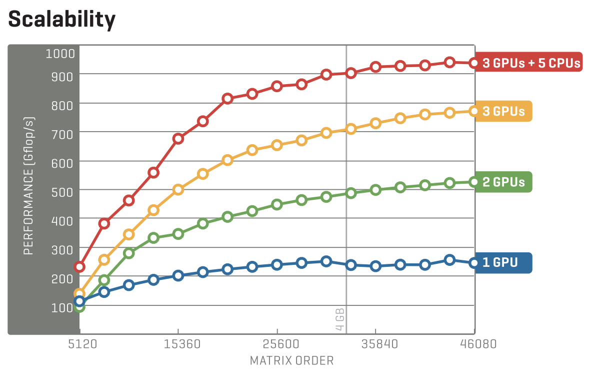


Figure 7: Performance scalability of the single precision Cholesky factorization.

supplementary cores allows to achieve 900 Gflop/s. The gap (120 Gflop/s) is higher than the potential of those five CPU cores since their cumulative *sgemm* peak is bounded by 100 Gflop/s. Although this result is non-intuitive, it can be explained as follows. GPUs are very efficient on regular level-3 BLAS computations such as *sgemm* (333 Gflop/s) but not that much on irregular level-2 BLAS kernels such as *spotrf* (56 Gflop/s). When both CPUs and GPUs are available, the CPUs can run most of the *spotrf* instances so that GPUs execute almost only *sgemm* operations. StarPU, being able to detect on the fly that property, schedules 80% of the *spotrf* on CPUs and dedicates GPUs for running *sgemm* whenever it can. Furthermore, the method transparently handles cases where the whole matrix does not fit in the memory of a GPU (when the matrix is larger than 4 GB, see Figure 7). Indeed, the runtime system simply moves back and forth parts of the data from the GPU memory to the CPU main memory, taking care of its coherency as it would do it between two different GPUs. Having a strong impact on the overall performance, the amount of data movement is automatically kept as low as possible according to the data-aware algorithm mentioned in Section 3.3. Figure 8 shows the impact of the data-aware policy: for large matrices, the total amount of data transfers is reduced more than twice, which results in an overall speed improvement of 25%.

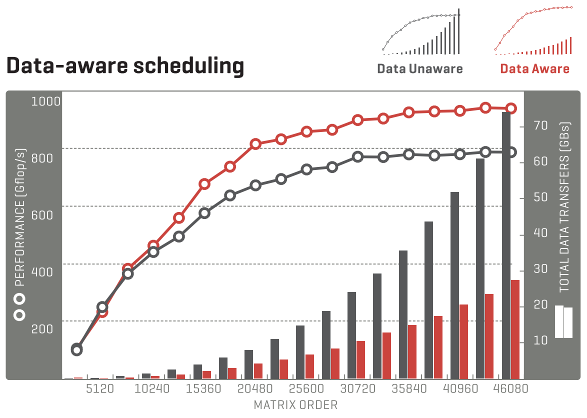


Figure 8: Impact of the data-aware policy on the amount of data transfers and on performance for the single precision Cholesky factorization.

5 Future Directions

The hybridization methodology presented in this chapter in the case of the Cholesky factorization has been successfully applied to a number of fundamental linear algebra algorithms [7, 8] for a single GPU. For the case of multi-GPUs however, some numerical algorithms (such as two-sided factorizations) are more complex to efficiently split into tasks of fine granularity that can be run concurrently. Therefore, the high-level re-design of numerical algorithms remains a challenging work for many applications. A similar methodology can be applied to execute algorithms on clusters of multicore nodes accelerated with GPUs. However, several major bottlenecks need to be alleviated to run at scale; this is an intensive research topic [9]. Finally, when a complex algorithm (such as Communication-Avoiding QR [10]) needs to be executed on a complex machine (such as an heterogeneous accelerated-based cluster), scheduling decisions may have a dramatic impact on performance. Therefore, new scheduling strategies will have to be designed to fully benefit from the potential of those future large-scale machines. One of the key concept will be to keep a strong interaction between the application and the runtime system in order to provide the opportunity to the scheduler to take the most appropriate decisions.

References

[1] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma, November 2009.

[2] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010.

[3] http://software.intel.com/en-us/intel-mkl/.

[4] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009.

[5] E. S. Quintana-Ortí and R. A. van de Geijn. Updating an LU factorization with pivoting. *ACM Trans. Math. Softw.*, 35(2):11, 2008.

[6] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[7] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. *Proc. of IPDPSW'10.*

[8] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, In Press, Corrected Proof, 2010.

[9] G. Bosilca, A. Bouteiller, A Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. *Innovative Computing Laboratory Technical Report.*

[10] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tile QR Factorization with Parallel Panel Processing for Multicore Architectures. *24th IEEE IPDPS*, 2010.