Chapter 1
Accelerating Numerical Dense Linear Algebra Calculations with GPUs

Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki

1.1 Introduction

Enabling large scale use of GPU-based architectures for high performance computational science depends on the successful development of fundamental numerical libraries for GPUs. Of particular interest are libraries in the area of dense linear algebra (DLA), as many science and engineering applications depend on them; these applications will not perform well unless the linear algebra libraries perform well.

Drivers for DLA developments have been significant hardware changes. In particular, the development of LAPACK [1]—the contemporary library for DLA computations—was motivated by the hardware changes in the late 1980s when its predecessors (EISPACK and LINPACK) needed to be redesigned to run efficiently on shared-memory vector and parallel processors with multilayered memory hierarchies. Memory hierarchies enable the caching of data for its reuse in computations, while reducing its movement. To account for this, the main DLA algorithms were reorganized to use block matrix operations, such as matrix multiplication, in their innermost loops. These block operations can be optimized for various architectures to account for memory hierarchy, and so provide a way to achieve high-efficiency on diverse architectures.

J. Dongarra
University of Tennessee Knoxville, Knoxville, TN 37996-3450, USA

Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA

University of Manchester, Manchester M13 9PL, UK
e-mail: dongarra@eecs.utk.edu

M. Gates • A. Haidar • J. Kurzak • P. Luszczek • S. Tomov (✉) • I. Yamazaki
University of Tennessee Knoxville, Knoxville, TN 37996-3450, USA
e-mail: mgates3@eecs.utk.edu; haidar@eecs.utk.edu; kurzak@eecs.utk.edu; luszczek@eecs.utk.edu; tomov@eecs.utk.edu; iyamazak@eecs.utk.edu

V. Kindratenko (ed.), *Numerical Computations with GPUs*,
DOI 10.1007/978-3-319-06548-9_1,

Challenges for DLA on GPUs stem from present-day hardware changes that require yet another major redesign of DLA algorithms and software in order to be efficient on modern architectures. This is provided through the MAGMA library [12], a redesign for GPUs of the popular LAPACK.

There are two main hardware trends that challenge and motivate the development of new algorithms and programming models, namely:

The explosion of parallelism where a single GPU can have thousands of cores (e.g., there are 2,880 CUDA cores in a K40), and algorithms must account for this level of parallelism in order to use the GPUs efficiently;

The growing gap of compute vs. data-movement capabilities that has been increasing exponentially over the years. To use modern architectures efficiently new algorithms must be designed to reduce their data movements. Current discrepancies between the compute- vs. memory-bound computations can be orders of magnitude, e.g., a K40 achieves about 1,240 Gflop/s on dgemm but only about 46 Gflop/s on dgemv.

This chapter presents the current best design and implementation practices that tackle the above mentioned challenges in the area of DLA. Examples are given with fundamental algorithms—from the matrix–matrix multiplication kernel written in CUDA (in Sect. 1.2) to the higher level algorithms for solving linear systems (Sects. 1.3 and 1.4), to eigenvalue and SVD problems (Sect. 1.5).

The complete implementations and more are available through the MAGMA library.[1] Similar to LAPACK, MAGMA is an open source library and incorporates the newest algorithmic developments from the linear algebra community.

1.2 BLAS

The *Basic Linear Algebra Subroutines* (BLAS) are the main building blocks for dense matrix software packages. The matrix multiplication routine is the most common and most performance-critical BLAS routine. This section presents the process of building a fast matrix multiplication GPU kernel in double precision, real arithmetic (dgemm), using the process of autotuning. The target is the Nvidia K40c card.

In the canonical form, matrix multiplication is represented by three nested loops (Fig. 1.1). The primary tool in optimizing matrix multiplication is the technique of loop tiling. Tiling replaces one loop with two loops: the inner loop incrementing the loop counter by one, and the outer loop incrementing the loop counter by the tiling factor. In the case of matrix multiplication, tiling replaces the three loops of Fig. 1.1 with the six loops of Fig. 1.2. Tiling of matrix multiplication exploits the *surface to volume effect*, i.e., execution of $O(n^3)$ floating-point operations over $O(n^2)$ data.

[1] http://icl.cs.utk.edu/magma/.

```
for (m = 0; m< M; m++)
  for (n = 0; n < N; n++)
    for (k = 0; k< K; k++)
      C[n][m] += A[k][m]*B[n][k];
```

Fig. 1.1 Canonical form of matrix multiplication

```
for (m_ = 0; m_< M; m_+=tileM)
 for (n_ = 0; n_ < N; n_+=tileN)
  for (k_ = 0; k_ < K; k_+=tileK)
   for (m = 0; m< tileM; m++)
    for (n = 0; n< tileN; n++)
     for (k = 0; k< tileK; k++)
      C[n_+n][m_+n] +=
       A[k_+k][m_+m]*
       B[n_+n][k_+k];
```

Fig. 1.2 Matrix multiplication with loop tiling

```
for (m_ = 0; m_< M; m_+=tileM)
 for (n_ = 0; n_ < N; n_+=tileN)
  for (k_ = 0; k_ < K; k_+=tileK)
  {
   instruction
   instruction
   instruction
    ...
  }
```

Fig. 1.3 Matrix multiplication with complete unrolling of tile operations

Next, the technique of loop unrolling is applied, which replaces the three innermost loops with a single block of straight-line code (a single *basic block*), as shown in Fig. 1.3. The purpose of unrolling is twofold: to reduce the penalty of looping (the overhead of incrementing loop counters, advancing data pointers and branching), and to increase instruction-level parallelism by creating sequences of independent instructions, which can fill out the processor's pipeline.

This optimization sequence is universal for almost any computer architecture, including "standard" superscalar processors with cache memories, as well as GPU accelerators and other less conventional architectures. Tiling, also referred to as blocking, is often applied at multiple levels, e.g., L2 cache, L1 cache, registers file, etc.

In the case of a GPU, the C matrix is overlaid with a 2D grid of thread blocks, each one responsible for computing a single tile of C. Since the code of a GPU kernel spells out the operation of a single thread block, the two outer loops disappear, and only one loop remains—the loop advancing along the k dimension, tile by tile.

Figure 1.4 shows the GPU implementation of matrix multiplication at the device level. Each thread block computes a tile of C (dark gray) by passing through a stripe of A and a stripe of B (light gray). The code iterates over A and B in chunks of K_{blk} (dark gray). The thread block follows the cycle of:

- making texture reads of the small, dark gray, stripes of A and B and storing them in shared memory,
- synchronizing threads with the `__syncthreads()` call,
- loading A and B from shared memory to registers and computing the product,
- synchronizing threads with the `__syncthreads()` call.

After the light gray stripes of A and B are completely swept, the tile of C is read, updated and stored back to device memory. Figure 1.5 shows closer what happens in the inner loop. The light gray area shows the shape of the thread block. The dark gray regions show how a single thread iterates over the tile.

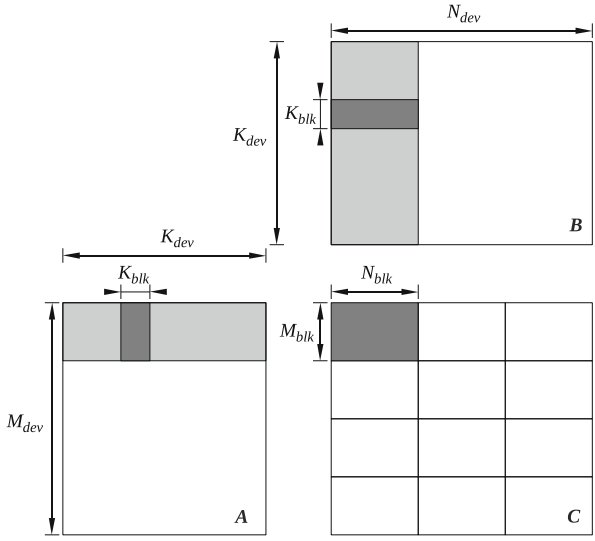

Fig. 1.4 gemm at the device level

Figure 1.6 shows the complete kernel implementation in CUDA. Tiling is defined by `BLK_M`, `BLK_N`, and `BLK_K`. `DIM_X` and `DIM_Y` define how the thread block covers the tile of C, `DIM_XA` and `DIM_YA` define how the thread block covers a stripe of A, and `DIM_XB` and `DIM_YB` define how the thread block covers a stripe of B.

In lines 24–28 the values of C are set to zero. In lines 32–38 a stripe of A is read (texture reads) and stored in shared memory. In lines 40–46 a stripe of B is read (texture reads) and stored in shared memory. The `__syncthreads()` call in line

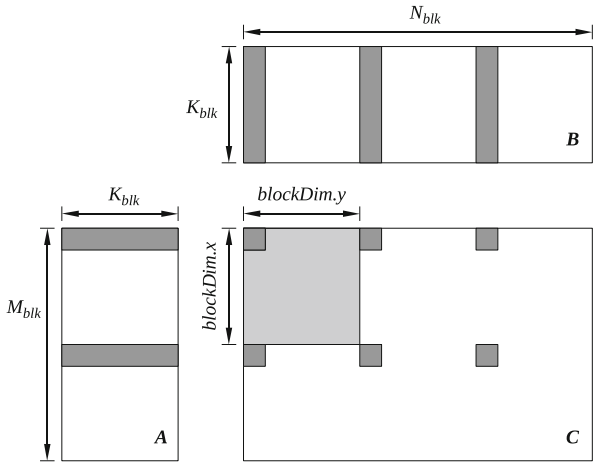

Fig. 1.5 gemm at the block level

48 ensures that reading of A and B, and storing in shared memory, is finished before operation continues. In lines 50–56 the product is computed, using the values from shared memory. The `__syncthreads()` call in line 58 ensures that computing the product is finished and the shared memory can be overwritten with new stripes of A and B. In lines 60 and 61 the pointers are advanced to the location of new stripes. When the main loop completes, C is read from device memory, modified with the accumulated product, and written back, in lines 64–77. The use of texture reads with clamping eliminates the need for *cleanup code* to handle matrix sizes not exactly divisible by the tiling factors.

With the parametrized code in place, what remains is the actual autotuning part, i.e., finding good values for the nine tuning parameters. Here the process used in the BEAST project (*Bench-testing Environment for Automated Software Tuning*) is described. It relies on three components: (1) defining the search space, (2) pruning the search space by applying filtering constraints, (3) benchmarking the remaining configurations and selecting the best performer. The important point in the BEAST project is to not introduce artificial, arbitrary limitations to the search process.

The loops of Fig. 1.7 define the search space for the autotuning of the matrix multiplication of Fig. 1.6. The two outer loops sweep through all possible 2D shapes of the thread block, up to the device limit in each dimension. The three inner loops sweep through all possible tiling sizes, up to arbitrarily high values, represented by the INF symbol. In practice, the actual values to substitute the INF symbols can be found by choosing a small starting point, e.g., (64, 64, 8), and moving up until further increase has no effect on the number of kernels that pass the selection.

The list of pruning constraints consists of nine simple checks that eliminate kernels deemed inadequate for one of several reasons:

- The kernel would not compile due to exceeding a hardware limit.
- The kernel would compile but fail to launch due to exceeding a hardware limit.

```
extern "C" __global__
void beast_gemm_kernel(
    int M, int N, int K,
    double alpha, double *A, int lda,
                  double *B, int ldb,
    double  beta, double *C, int ldc )
{
    int blx = blockIdx.x;        // block's m position
    int bly = blockIdx.y;        // block's n position
    int idx = threadIdx.x;       // thread's m position in C
    int idy = threadIdx.y;       // thread's n position in C
    int idt = DIM_X*idy+idx;     // thread's number

    int idxA = idt % DIM_XA;     // thread's m position for loading A
    int idyA = idt / DIM_XA;     // thread's n position for loading A
    int idxB = idt % DIM_XB;     // thread's m position for loding B
    int idyB = idt / DIM_XB;     // thread's n position for loading B

    __shared__ double sA[BLK_K][BLK_M+1];   // shared memory buffer for A
    __shared__ double sB[BLK_N][BLK_K+1];   // shared memory buffer for B
    double rC[BLK_N/DIM_Y][BLK_M/DIM_X];    // registers for C

    int coord_A = blx*BLK_M     + idyA*lda+idxA;    // A stripe's initial location
    int coord_B = bly*BLK_N*ldb + idyB*ldb+idxB;    // B stripe's initial location
    int m, n, k, kk;                                // loop counters

    #pragma unroll
    for (n = 0; n < BLK_N/DIM_Y; n++)
      #pragma unroll
      for (m = 0; m < BLK_M/DIM_X; m++)
        rC[n][m] = 0.0;

    for (kk = 0; kk < K; kk += BLK_K)
    {
        #pragma unroll
        for (n = 0; n < BLK_K; n += DIM_YA)
            #pragma unroll
            for (m = 0; m < BLK_M; m += DIM_XA) {
                int2 v = tex1Dfetch(tex_ref_A, coord_A + n*lda+m);
                sA[n+idyA][m+idxA] = __hiloint2double(v.y, v.x);
            }

        #pragma unroll
        for (n = 0; n < BLK_N; n += DIM_YB)
            #pragma unroll
            for (m = 0; m < BLK_K; m += DIM_XB) {
                int2 v = tex1Dfetch(tex_ref_B, coord_B + n*ldb+m);
                sB[n+idyB][m+idxB] = __hiloint2double(v.y, v.x);
            }

        __syncthreads();

        #pragma unroll
        for (k = 0; k < BLK_K; k++)
            #pragma unroll
            for (n = 0; n < BLK_N/DIM_Y; n++)
                #pragma unroll
                for (m = 0; m < BLK_M/DIM_X; m++)
                    rC[n][m] += sA[k][m*DIM_X+idx] * sB[n*DIM_Y+idy][k];

        __syncthreads();

        coord_A += BLK_K*lda;
        coord_B += BLK_K;
    }

    #pragma unroll
    for (n = 0; n < BLK_N/DIM_Y; n++) {
        int coord_dCn = bly*BLK_N + n*DIM_Y+idy;
        #pragma unroll
        for (m = 0; m < BLK_M/DIM_X; m++) {
            int coord_dCm = blx*BLK_M + m*DIM_X+idx;
            if (coord_dCm < M && coord_dCn < N) {
                int offsC = coord_dCn*ldc + coord_dCm;
                double &regC = rC[n][m];
                double &memC = C[offsC];
                memC = alpha*regC + beta*memC;
            }
        }
    }
}
```

Fig. 1.6 Complete dgemm (C = alpha A B + beta C) implementation in CUDA

```
1  // Sweep thread block dimensions.
2  for (dim_m = 1; dim_m <=MAX_THREADS_DIM_X; dim_m++)
3   for (dim_n = 1; dim_n <=MAX_THREADS_DIM_Y; dim_n++)
4    // Sweep tiling sizes.
5    for (blk_m = dim_m; blk_m < INF; blk_m += dim_m)
6     for (blk_n = dim_n; blk_n < INF; blk_n += dim_n)
7      for (blk_k = 1; blk_k < INF; blk_k++)
8      {
9       // Apply pruning constraints.
10     }
```

Fig. 1.7 The parameter search space for the autotuning of matrix multiplication

- The kernel would compile and launch, but produce invalid results due to the limitations of the implementation, e.g., unimplemented corner case.
- The kernel would compile, launch and produce correct results, but have no chance of running fast, due to an obvious performance shortcoming, such as very low occupancy.

The nine checks rely on basic hardware parameters, which can be obtained by querying the card with the CUDA API, and include:

1. The number of threads in the block is not divisible by the warp size.
2. The number of threads in the block exceeds the hardware maximum.
3. The number of registers per thread, to store C, exceeds the hardware maximum.
4. The number of registers per block, to store C, exceeds the hardware maximum.
5. The shared memory per block, to store A and B, exceeds the hardware maximum.
6. The thread block cannot be shaped to read A and B without cleanup code.
7. The number of load instructions, from shared memory to registers, in the innermost loop, in the PTX code, exceeds the number of *Fused Multiply-Adds* (FMAs).
8. Low occupancy due to high number of registers per block to store C.
9. Low occupancy due to the amount of shared memory per block to read A and B.

In order to check the last two conditions, the number of registers per block, and the amount of shared memory per block are computed. Then the maximum number of possible blocks per multiprocessor is found, which gives the maximum possible number of threads per multiprocessor. If that number is lower than the minimum occupancy requirement, the kernel is discarded. Here the threshold is set to a fairly low number of 256 threads, which translates to minimum occupancy of 0.125 on the Nvidia K40 card, with the maximum number of 2,048 threads per multiprocessor.

This process produces 14,767 kernels, which can be benchmarked in roughly 1 day. Three thousand two hundred and fifty six kernels fail to launch due to excessive number of registers per block. The reason is that the pruning process uses a lower estimate on the number of registers, and the compiler actually produces code requiring more registers. We could detect it in compilation and skip benchmarking

of such kernels or we can run them and let them fail. For simplicity we chose the latter. We could also cap the register usage to prevent the failure to launch. However, capping register usage usually produces code of inferior performance.

Eventually, 11,511 kernels run successfully and pass correctness checks. Figure 1.8 shows the performance distribution of these kernels. The fastest kernel achieves 900 Gflop/s with tiling of $96 \times 64 \times 12$, with 128 threads (16×8 to compute C, 32×4 to read A, and 4×32 to read B). The achieved occupancy number of 0.1875 indicates that, most of the time, each multiprocessor executes 384 threads (three blocks).

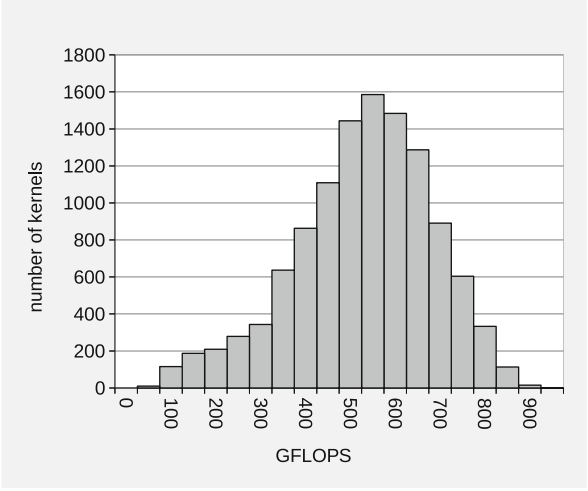

Fig. 1.8 Distribution of the dgemm kernels

In comparison, CUBLAS achieves the performance of 1,225 Gflop/s using 256 threads per multiprocessor. Although CUBLAS achieves a higher number, this example shows the effectiveness of the autotuning process in quickly creating well performing kernels from high level language source codes. This technique can be used to build kernels for routines not provided in vendor libraries, such as extended precision BLAS (double–double and triple-float), BLAS for misshaped matrices (tall and skinny), etc. Even more importantly, this technique can be used to build domain specific kernels for many application areas.

As the last interesting observation, we offer a look at the PTX code produced by the `nvcc` compiler (Fig. 1.9). We can see that the compiler does exactly what is expected, which is completely unrolling the loops in lines 50–56 of the C code in Fig. 1.6, into a stream of loads from shared memory to registers and FMA instructions, with substantially more FMAs than loads.

```
ld.shared.f64  %fd258, [%rd3];
ld.shared.f64  %fd259, [%rd4];
fma.rn.f64  %fd260, %fd258, %fd259, %fd1145;
ld.shared.f64  %fd261, [%rd3+128];
fma.rn.f64  %fd262, %fd261, %fd259, %fd1144;
ld.shared.f64  %fd263, [%rd3+256];
fma.rn.f64  %fd264, %fd263, %fd259, %fd1143;
ld.shared.f64  %fd265, [%rd3+384];
fma.rn.f64  %fd266, %fd265, %fd259, %fd1142;
ld.shared.f64  %fd267, [%rd3+512];
fma.rn.f64  %fd268, %fd267, %fd259, %fd1141;
ld.shared.f64  %fd269, [%rd3+640];
fma.rn.f64  %fd270, %fd269, %fd259, %fd1140;
ld.shared.f64  %fd271, [%rd4+832];
fma.rn.f64  %fd272, %fd258, %fd271, %fd1139;
fma.rn.f64  %fd273, %fd261, %fd271, %fd1138;
fma.rn.f64  %fd274, %fd263, %fd271, %fd1137;
fma.rn.f64  %fd275, %fd265, %fd271, %fd1136;
fma.rn.f64  %fd276, %fd267, %fd271, %fd1135;
fma.rn.f64  %fd277, %fd269, %fd271, %fd1134;
ld.shared.f64  %fd278, [%rd4+1664];
fma.rn.f64  %fd279, %fd258, %fd278, %fd1133;
fma.rn.f64  %fd280, %fd261, %fd278, %fd1132;
fma.rn.f64  %fd281, %fd263, %fd278, %fd1131;
fma.rn.f64  %fd282, %fd265, %fd278, %fd1130;
fma.rn.f64  %fd283, %fd267, %fd278, %fd1129;
fma.rn.f64  %fd284, %fd269, %fd278, %fd1128;
ld.shared.f64  %fd285, [%rd4+2496];
fma.rn.f64  %fd286, %fd258, %fd285, %fd1127;
fma.rn.f64  %fd287, %fd261, %fd285, %fd1126;
fma.rn.f64  %fd288, %fd263, %fd285, %fd1125;
fma.rn.f64  %fd289, %fd265, %fd285, %fd1124;
fma.rn.f64  %fd290, %fd267, %fd285, %fd1123;
fma.rn.f64  %fd291, %fd269, %fd285, %fd1122;
ld.shared.f64  %fd292, [%rd4+3328];
fma.rn.f64  %fd293, %fd258, %fd292, %fd1121;
fma.rn.f64  %fd294, %fd261, %fd292, %fd1120;
fma.rn.f64  %fd295, %fd263, %fd292, %fd1119;
fma.rn.f64  %fd296, %fd265, %fd292, %fd1118;
fma.rn.f64  %fd297, %fd267, %fd292, %fd1117;
fma.rn.f64  %fd298, %fd269, %fd292, %fd1116;
ld.shared.f64  %fd299, [%rd4+4160];
fma.rn.f64  %fd300, %fd258, %fd299, %fd1115;
fma.rn.f64  %fd301, %fd261, %fd299, %fd1114;
fma.rn.f64  %fd302, %fd263, %fd299, %fd1113;
fma.rn.f64  %fd303, %fd265, %fd299, %fd1112;
fma.rn.f64  %fd304, %fd267, %fd299, %fd1111;
fma.rn.f64  %fd305, %fd269, %fd299, %fd1110;
ld.shared.f64  %fd306, [%rd4+4992];
fma.rn.f64  %fd307, %fd258, %fd306, %fd1109;
fma.rn.f64  %fd308, %fd261, %fd306, %fd1108;
fma.rn.f64  %fd309, %fd263, %fd306, %fd1107;
fma.rn.f64  %fd310, %fd265, %fd306, %fd1106;
fma.rn.f64  %fd311, %fd267, %fd306, %fd1105;
fma.rn.f64  %fd312, %fd269, %fd306, %fd1104;
ld.shared.f64  %fd313, [%rd4+5824];
fma.rn.f64  %fd314, %fd258, %fd313, %fd1103;
fma.rn.f64  %fd315, %fd261, %fd313, %fd1102;
fma.rn.f64  %fd316, %fd263, %fd313, %fd1101;
fma.rn.f64  %fd317, %fd265, %fd313, %fd1100;
fma.rn.f64  %fd318, %fd267, %fd313, %fd1099;
fma.rn.f64  %fd319, %fd269, %fd313, %fd1098;
ld.shared.f64  %fd320, [%rd3+776];
ld.shared.f64  %fd321, [%rd4+8];
fma.rn.f64  %fd322, %fd320, %fd321, %fd260;
ld.shared.f64  %fd323, [%rd3+904];
fma.rn.f64  %fd324, %fd323, %fd321, %fd262;
ld.shared.f64  %fd325, [%rd3+1032];
fma.rn.f64  %fd326, %fd325, %fd321, %fd264;
ld.shared.f64  %fd327, [%rd3+1160];
fma.rn.f64  %fd328, %fd327, %fd321, %fd266;
ld.shared.f64  %fd329, [%rd3+1288];
fma.rn.f64  %fd330, %fd329, %fd321, %fd268;
ld.shared.f64  %fd331, [%rd3+1416];
fma.rn.f64  %fd332, %fd331, %fd321, %fd270;
ld.shared.f64  %fd333, [%rd4+840];
fma.rn.f64  %fd334, %fd320, %fd333, %fd272;
fma.rn.f64  %fd335, %fd323, %fd333, %fd273;
fma.rn.f64  %fd336, %fd325, %fd333, %fd274;
fma.rn.f64  %fd337, %fd327, %fd333, %fd275;
fma.rn.f64  %fd338, %fd329, %fd333, %fd276;
fma.rn.f64  %fd339, %fd331, %fd333, %fd277;
ld.shared.f64  %fd340, [%rd4+1672];
fma.rn.f64  %fd341, %fd320, %fd340, %fd279;
fma.rn.f64  %fd342, %fd323, %fd340, %fd280;
fma.rn.f64  %fd343, %fd325, %fd340, %fd281;
fma.rn.f64  %fd344, %fd327, %fd340, %fd282;
fma.rn.f64  %fd345, %fd329, %fd340, %fd283;
fma.rn.f64  %fd346, %fd331, %fd340, %fd284;
```

Fig. 1.9 A portion of the PTX for the innermost loop of the fastest dgemm kernel

1.3 Solving Linear Systems

Solving dense linear systems of equations is a fundamental problem in scientific computing. Numerical simulations involving complex systems represented in terms of unknown variables and relations between them often lead to linear systems of equations that must be solved as fast as possible. This section presents a methodology for developing these solvers. The technique is illustrated using the Cholesky factorization.

1.3.1 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 [5]. 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.

1.3.2 Hybrid Algorithms

The Cholesky factorization 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). The number of synchronizations of this algorithm and the synchronous nature of the panel factorization would be prohibitive bottlenecks for performance on highly parallel devices such as GPUs.

Instead, the panel factorization and the update of the trailing submatrix are broken into tasks, where the less parallel panel tasks are scheduled for execution on multicore CPUs, and the parallel updates mainly on GPUs. Figure 1.10 illustrates this concept of developing hybrid algorithms by splitting the computation into tasks, data dependencies, and consequently scheduling the execution over GPUs and multicore CPUs. The scheduling can be static (described next), or dynamic (see Sect. 1.4). In either case, the small and not easy to parallelize tasks from the critical path (e.g., panel factorizations) are executed on CPUs, and the large and highly parallel task (like the matrix updates) are executed mostly on the GPUs.

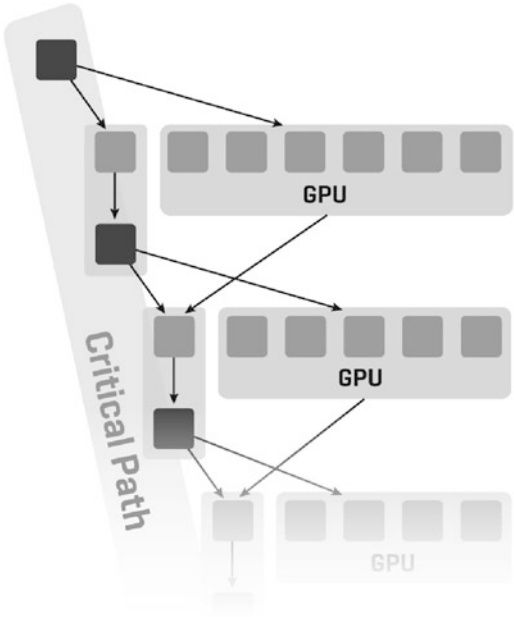

Fig. 1.10 Algorithms as a collection of tasks and dependencies among them for hybrid GPU-CPU computing

1.3.3 Hybrid Cholesky Factorization for a Single GPU

Figure 1.11 gives the hybrid Cholesky factorization implementation for a single GPU. Here `da` points to the input matrix that is in the GPU memory, `work` is a work-space array in 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

```
1  for (j = 0; j < *n; j += nb){
2    jb = min(nb, *n-j);
3    cublasDsyrk('l','n', jb, j,-1, da(j,0),*lda, 1, da(j,j),*lda);
4    cudaMemcpy2DAsync(work, jb*sizeof(double), da(j,j),*lda*sizeof(double),
5                      sizeof(double)*jb, jb, cudaMemcpyDeviceToHost, stream[1]);
6    if (j + jb < *n)
7      cublasDgemm('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   dpotrf_("Lower", &jb, work, &jb, info);
11   if (*info != 0)
12     *info = *info + j, break;
13   cudaMemcpy2DAsync(da(j,j), *lda*sizeof(double), work, jb*sizeof(double),
14                     sizeof(double)*jb, jb, cudaMemcpyHostToDevice, stream[0]);
15   if (j + jb < *n)
16     cublasDtrsm('r','l','t','n', *n-j-jb, jb, 1, da(j,j),*lda,
17                 da(j+jb,j),*lda);
18 }
```

Fig. 1.11 Hybrid Cholesky factorization for single CPU-GPU pair (*dpotrf*)

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 ensure that the data has arrived, then factor it on the CPU using a 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 statically scheduling first the *dgemm* (line 7) on the GPU; this is an asynchronous call, hence the CPU continues immediately with the *dpotrf* (line 10) while the GPU is running the *dgemm*.

The hybrid algorithm is given an LAPACK interface to simplify its use and adoption. Thus, codes that use LAPACK can be seamlessly accelerated multiple times with GPUs.

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 assigned for execution on the GPU;
- Small, inherently sequential *dpotrf* 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, $j \geq nb$.

1.4 The Case for Dynamic Scheduling

In this section, we present the linear algebra aspects of our generic solution for development of either Cholesky, Gaussian, and Householder factorizations based on block outer-product updates of the trailing matrix.

Conceptually, one-sided factorization $\mathscr{F}$ maps a matrix A into a product of two matrices X and Y:

$$\mathscr{F} : \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mapsto \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \times \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}$$

Algorithmically, this corresponds to a sequence of in-place transformations of A, whose storage is overwritten with the entries of matrices X and Y (P_{ij} indicates the currently factorized panels):

$$\begin{bmatrix} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(0)} \\ A_{21}^{(0)} & A_{22}^{(0)} & A_{23}^{(0)} \\ A_{31}^{(0)} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} P_{11} & A_{12}^{(0)} & A_{13}^{(0)} \\ P_{21} & A_{22}^{(0)} & A_{23}^{(0)} \\ P_{31} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & A_{22}^{(1)} & A_{23}^{(1)} \\ X_{31} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & P_{22} & A_{23}^{(1)} \\ X_{31} & P_{32} & A_{33}^{(1)} \end{bmatrix} \rightarrow$$

Algorithm 1 Two-phase implementation of a one-sided factorization

```
// iterate over all matrix panels
for P_i ∈ {P_1, P_2, ..., P_n}
    FactorizePanel(P_i)
    UpdateTrailingMatrix(A^(i))
end
```

Table 1.1 Routines for panel factorization and the trailing matrix update

	Cholesky	Householder	Gauss
FactorizePanel	dpotf2	dgeqf2	dgetf2
	dtrsm		
	dsyrk	dlarfb	dlaswp
UpdateTrailingMatrix	dgemm		dtrsm
			dgemm

Algorithm 2 Two-phase implementation with the update split between Fermi and Kepler GPUs

```
// iterate over all matrix panels
for P_i ∈ {P_1, P_2, ...}
    FactorizePanel(P_i)
    UpdateTrailingMatrix_Kepler(A^(i))
    UpdateTrailingMatrix_Fermi(A^(i))
end
```

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & A_{33}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & X_{22} & Y_{23} \\ X_{31} & X_{32} & P_{33} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & XY_{33} \end{bmatrix} \rightarrow [XY],$$

where XY_{ij} is a compact representation of both X_{ij} and Y_{ij} in the space originally occupied by A_{ij}.

Observe two distinct phases in each step of the transformation from $[A]$ to $[XY]$: *panel factorization* (P) and trailing matrix update: $A^{(i)} \rightarrow A^{(i+1)}$. Implementation of these two phases leads to a straightforward iterative scheme shown in Algorithm 1. Table 1.1 shows BLAS and LAPACK routines that should be substituted for the generic routines named in the algorithm.

The use of multiple accelerators complicates the simple loop from Algorithm 1: we must split the update operation into multiple instances for each of the accelerators. This was done in Algorithm 2. Notice that FactorizePanel() is not split for execution on accelerators because it exhibits properties of latency-bound workloads, which face a number of inefficiencies on throughput-oriented GPU devices. Due to their high performance rate exhibited on the update operation, and the fact that the update requires the majority of floating-point operations, it is the trailing matrix update that is a good target for off-load. The problem of keeping track of the computational activities is exacerbated by the separation between the address spaces of main memory of the CPU and the GPUs. This requires synchronization between memory buffers and is included in the implementation shown in Algorithm 3.

Algorithm 3 Two-phase implementation with a split update and explicit communication

```
// iterate over all matrix panels
for Pi ∈ {P1, P2, ...}
    FactorizePanel(Pi)
    SendPanel_Kepler(Pi)
    UpdateTrailingMatrix_Kepler(A^(i))
    SendPanel_Fermi(Pi)
    UpdateTrailingMatrix_Fermi(A^(i))
end
```

Algorithm 4 Lookahead of depth 1 for the two-phase factorization

```
FactorizePanel(P1)
SendPanel(P1)
UpdateTrailingMatrix_{Kepler, Fermi}(P1)
PanelStartReceiving(P2)
UpdateTrailingMatrix_{Kepler, Fermi}(R^(1))
// iterate over remaining matrix panels
for Pi ∈ {P2, P3, ...}
    PanelReceive(Pi)
    PanelFactor(Pi)
    SendPanel(Pi)
    UpdateTrailingMatrix_{Kepler, Fermi}(Pi)
    PanelStartReceiving(Pi)
    UpdateTrailingMatrix_{Kepler, Fermi}(R^(i))
end
PanelReceive(Pn)
PanelFactor(Pn)
```

The complexity increases further as the code must be modified further to achieve close to peak performance. In fact, the bandwidth between the CPU and the GPUs is orders of magnitude too slow to sustain computational rates of GPUs.[2] The common technique to alleviate this imbalance is to use *lookahead* [14, 15].

Algorithm 4 shows a very simple case of a lookahead of depth 1. The update operation is split into an update of the next panel, the start of the receiving of the next panel that just got updated, and an update of the rest of the trailing matrix R. The splitting is done to overlap the communication of the panel and the update operation. The complication of this approach comes from the fact that depending on the communication bandwidth and the accelerator speed, a different lookahead depth might be required for optimal overlap. In fact, the adjustment of the depth is often required throughout the factorization's runtime to yield good performance: the updates consume progressively less time when compared to the time spent in the panel factorization.

[2]The bandwidth for the current generation PCI Express is at most 16 GB/s while the devices achieve over 1,000 Gflop/s performance.

Since the management of adaptive lookahead is tedious, it is desirable to use a dynamic scheduler to keep track of data dependences and communication events. The only issue is the homogeneity inherent in most of the schedulers which is violated here due to the use of three different computing devices that we used. Also, common scheduling techniques, such as task stealing, are not applicable here due to the disjoint address spaces and the associated large overheads. These caveats are dealt with comprehensively in the remainder of the chapter.

1.5 Eigenvalue and Singular Value Problems

Eigenvalue and singular value decomposition (SVD) problems are fundamental for many engineering and physics applications. For example, image processing, compression, facial recognition, vibrational analysis of mechanical structures, and computing energy levels of electrons in nanostructure materials can all be expressed as eigenvalue problems. Also, the SVD plays a very important role in statistics where it is directly related to the principal component analysis method, in signal processing and pattern recognition as an essential filtering tool, and in analysis of control systems. It has applications in such areas as least squares problems, computing the pseudoinverse, and computing the Jordan canonical form. In addition, the SVD is used in solving integral equations, digital image processing, information retrieval, seismic reflection tomography, and optimization.

1.5.1 Background

The eigenvalue problem is to find an eigenvector x and eigenvalue λ that satisfy

$$Ax = \lambda x,$$

where A is a symmetric or nonsymmetric $n \times n$ matrix. When the entire eigenvalue decomposition is computed we have $A = X\Lambda X^{-1}$, where Λ is a diagonal matrix of eigenvalues and X is a matrix of eigenvectors. The SVD finds orthogonal matrices U, V, and a diagonal matrix Σ with nonnegative elements, such that $A = U\Sigma V^T$, where A is an $m \times n$ matrix. The diagonal elements of Σ are singular values of A, the columns of U are called its left singular vectors, and the columns of V are called its right singular vectors.

All of these problems are solved by a similar three-phase process:

1. **Reduction phase**: orthogonal matrices Q (Q and P for singular value decomposition) are applied on both the left and the right side of A to reduce it to a condensed form matrix—hence these are called "two-sided factorizations." Note that the use of two-sided orthogonal transformations guarantees that A has the

same eigen/singular-values as the reduced matrix, and the eigen/singular-vectors of A can be easily derived from those of the reduced matrix (step 3);
2. **Solution phase**: an eigenvalue (respectively, singular value) solver further computes the eigenpairs Λ and Z (respectively, singular values Σ and the left and right vectors $\tilde{U}$ and $\tilde{V}^T$) of the condensed form matrix;
3. **Back transformation phase**: if required, the eigenvectors (respectively, left and right singular vectors) of A are computed by multiplying Z (respectively, $\tilde{U}$ and $\tilde{V}^T$) by the orthogonal matrices used in the reduction phase.

For the nonsymmetric eigenvalue problem, the reduction phase is to upper Hessenberg form, $H = Q^T A Q$. For the second phase, QR iteration is used to find the eigenpairs of the reduced Hessenberg matrix H by further reducing it to (quasi) upper triangular Schur form, $S = E^T H E$. Since S is in a (quasi) upper triangular form, its eigenvalues are on its diagonal and its eigenvectors Z can be easily derived. Thus, A can be expressed as:

$$A = QHQ^T = Q\ E\ S\ E^T\ Q^T,$$

which reveals that the eigenvalues of A are those of S, and the eigenvectors Z of S can be back-transformed to eigenvectors of A as $X = Q\ E\ Z$.

When A is symmetric (or Hermitian in the complex case), the reduction phase is to symmetric tridiagonal $T = Q^T A Q$, instead of upper Hessenberg form. Since T is tridiagonal, computations with T are very efficient. Several eigensolvers are applicable to the symmetric case, such as the divide and conquer (D&C), the multiple relatively robust representations (MRRR), the bisection algorithm, and the QR iteration method. These solvers compute the eigenvalues and eigenvectors of $T = Z \Lambda Z^T$, yielding Λ to be the eigenvalues of A. Finally, if eigenvectors are desired, the eigenvectors Z of T are back-transformed to eigenvectors of A as $X = Q\ Z$.

For the singular value decomposition (SVD), two orthogonal matrices Q and P are applied on the left and on the right, respectively, to reduce A to bidiagonal form, $B = Q^T A P$. Divide and conquer or QR iteration is then used as a solver to find both the singular values and the left and the right singular vectors of B as $B = \tilde{U} \Sigma \tilde{V}^T$, yielding the singular values of A. If desired, singular vectors of B are back-transformed to singular vectors of A as $U = Q\ \tilde{U}$ and $V^T = P^T \tilde{V}^T$.

There are many ways to formulate mathematically and solve these problems numerically, but in all cases, designing an efficient computation is challenging because of the nature of the algorithms. In particular, the orthogonal transformations applied to the matrix are two-sided, i.e., transformations are applied on both the left and right side of the matrix. This creates data dependencies that prevent the use of standard techniques to increase the computational intensity of the computation, such as blocking and look-ahead, which are used extensively in the one-sided LU, QR, and Cholesky factorizations. Thus, the reduction phase can take a large portion of the overall time. Recent research has been into two-stage algorithms [2, 6, 7, 10, 11], where the first stage uses Level 3 BLAS operations to reduce A

to band form, followed by a second stage to reduce it to the final condensed form. Because it is the most time consuming phase, it is very important to identify the bottlenecks of the reduction phase, as implemented in the classical approaches [1]. The classical approach is discussed in the next section, while Sect. 1.5.4 covers two-stage algorithms.

The initial reduction to condensed form (Hessenberg, tridiagonal, or bidiagonal) and the final back-transformation are particularly amenable to GPU computation. The eigenvalue solver itself (QR iteration or divide and conquer) has significant control flow and limited parallelism, making it less suited for GPU computation.

1.5.2 Classical Reduction to Hessenberg, Tridiagonal, or Bidiagonal Condensed Form

The classical approach (*"LAPACK algorithms"*) to reduce a matrix to condensed form is to use one-stage algorithms [5]. Similar to the one-sided factorizations (LU, Cholesky, QR), the two-sided factorizations are split into a *panel factorization* and a *trailing matrix update*. Pseudocode for the Hessenberg factorization is given in Algorithm 5 and shown schematically in Fig. 1.12; the tridiagonal and bidiagonal factorizations follow a similar form, though the details differ [17]. Unlike the one-sided factorizations, the panel factorization requires computing Level 2 BLAS matrix-vector products with the entire trailing matrix. This requires loading the entire trailing matrix into memory, incurring a significant amount of memory bound operations. It also produces synchronization points between the panel factorization and the trailing submatrix update steps. As a result, the algorithm follows the expensive fork-and-join model, preventing overlap between the CPU computation and the GPU computation. Also it prevents having a look-ahead panel and hiding communication costs by overlapping with computation. For instance, in the Hessenberg factorization, these Level 2 BLAS operations account for about 20 % of the floating point operations, but can take 70 % of the time in a CPU implementation [16]. Note that the computational complexity of the reduction phase is about $\frac{10}{3}n^3$, $\frac{8}{3}n^3$, and $\frac{4}{3}n^3$ for the reduction to Hessenberg, bidiagonal, and tridiagonal form respectively.

In the panel factorization, each column is factored by introducing zeros below the subdiagonal using an orthogonal Householder reflector, $H_j = I - \tau v_j v_j^T$. The matrix Q is represented as a product of $n-1$ of these reflectors,

$$Q = H_1 H_2 \ldots H_{n-1}.$$

Before the next column can be factored, it must be updated as if H_j were applied on both sides of A, though we delay actually updating the trailing matrix. For each column, performing this update requires computing $y_j = Av_j$. For a GPU implementation, we compute these matrix-vector products on the GPU, using cublasDgemv for the Hessenberg and bidiagonal, and cublasDsymv for the tridiagonal factorization. Optimized versions of symv and hemv also exist in

Algorithm 5 Hessenberg reduction, magma_*gehrd

for $i = 1, \ldots, n$ by nb
  // panel factorization, in magma_*lahr2.*
  get panel $A_{i:n,i:i+nb-1}$ from GPU
  for $j = i, \ldots, i + nb$
    $(v_j, \tau_j) =$ householder(a_j)
    send v_j to GPU
    $y_j = A_{i+1:n,j:n} v_j$ on GPU
    get y_j from GPU
    compute $T_{(j)} = \begin{bmatrix} T_{(j-1)} & -\tau_j T_{(j-1)} V_{(j-1)}^T v_j \\ 0 & \tau_j \end{bmatrix}$
    update column $a_{j+1} = (I - VT^T V^T)(a_{j+1} - YT\{V^T\}_{j+1})$
  end

  // trailing matrix update, in magma_*lahru.*
  $Y_{1:i,1:nb} = A_{1:i,i:n} V$ on GPU
  $A = (I - VT^T V^T)(A - YTV^T)$ on GPU
end

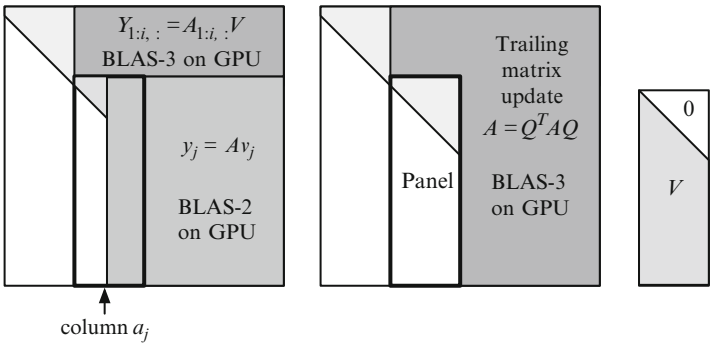

Fig. 1.12 Hessenberg panel factorization, trailing matrix update, and V matrix on GPU with upper triangle set to zero

MAGMA [13], which achieve higher performance by reading A only once and using extra workspace to store intermediate results. While these are memory-bound Level 2 BLAS operations, computing them on the GPU leverages the GPU's high memory bandwidth.

After factoring each panel of nb columns, the trailing matrix must be updated. Instead of applying each H_j individually to the entire trailing matrix, they are blocked together into a block Hessenberg update,

$$Q_i = H_1 H_2 \ldots H_{nb} = I - V_i T_i V_i^T.$$

The trailing matrix is then updated as

$$\hat{A} = Q_i^T A Q_i = (I - V_i T_i^T V_i^T)(A - Y_i T_i V_i^T) \tag{1.1}$$

for the nonsymmetric case, or using the alternate representation

$$\hat{A} = A - W_i V_i^T - V_i W_i^T \tag{1.2}$$

for the symmetric case. In all cases, the update is a series of efficient Level 3 BLAS operations executed on the GPU, either general matrix–matrix multiplies (dgemm) for the Hessenberg and bidiagonal factorizations, or a symmetric rank-$2k$ update (dsyr2k) for the symmetric tridiagonal factorization.

Several additional considerations are made for an efficient GPU implementation. In the LAPACK CPU implementation, the matrix V of Householder vectors is stored below the subdiagonal of A. This requires multiplies to be split into two operations, a triangular multiply (dtrmm) for the top triangular portion, and a dgemm for the bottom portion. On the GPU, we explicitly set the upper triangle of V to zero, as shown in Fig. 1.12, so the entire product can be computed using a single dgemm. Second, it is beneficial to store the small $nb \times nb$ T_i matrices used in the reduction, for later use in the back-transformation, whereas LAPACK recomputes them later from V_i.

1.5.3 Back-Transform Eigenvectors

For eigenvalue problems, after the reduction to condensed form, the eigensolver finds the eigenvalues Λ and eigenvectors Z of H or T. For the SVD, it finds the singular values Σ and singular vectors $\tilde{U}$ and $\tilde{V}$ of B. The eigenvalues and singular values are the same as for the original matrix A. To find the eigenvectors or singular vectors of the original matrix A, the vectors need to be back-transformed by multiplying by the same orthogonal matrix Q (and P, for the SVD) used in the reduction to condensed form. As in the reduction, the block Householder transformation $Q_i = I - V_i T_i V_i^T$ is used. From this representation, either Q can be formed explicitly using dorghr, dorgtr, or dorgbr; or we can multiply by the implicitly represented Q using dormhr, dormtr, or dormbr. In either case, applying it becomes a series of dgemm operations executed on the GPU.

The entire procedure is implemented in the MAGMA library: magma_dgeev for nonsymmetric eigenvalues, magma_dsyevd for real symmetric, and magma_dgesvd for the singular value decomposition.

1.5.4 Two Stage Reduction

Because of the expense of the reduction step, renewed research has focused on improving this step, resulting in a novel technique based on a two-stage reduction [6, 9]. The two-stage reduction is designed to increase the utilization of compute-intensive operations. Many algorithms have been investigated using this two-stage approach. The idea is to split the original one-stage approach into a compute-intensive phase (first stage) and a memory-bound phase (second or "bulge chasing" stage). In this section we will cover the description for the symmetric case. The first stage reduces the original symmetric dense matrix to a symmetric band form, while the second stage reduces from band to tridiagonal form, as depicted in Fig. 1.13.

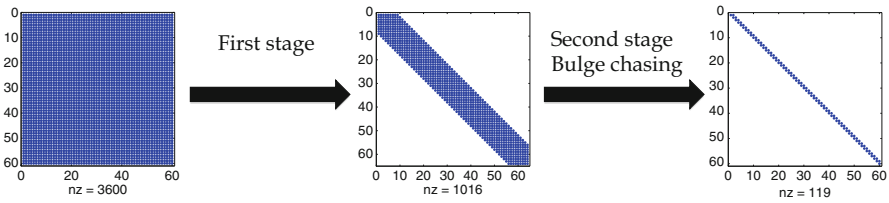

Fig. 1.13 Two stage technique for the reduction phase

1.5.4.1 First Stage: Hybrid CPU-GPU Band Reduction

The first stage applies a sequence of block Householder transformations to reduce a symmetric dense matrix to a symmetric band matrix. This stage uses compute-intensive matrix-multiply kernels, eliminating the memory-bound matrix-vector product in the one-stage panel factorization, and has been shown to have a good data access pattern and large portion of Level 3 BLAS operations [3, 4, 8]. It also enables the efficient use of GPUs by minimizing communication and allowing overlap of computation and communication. Given a dense $n \times n$ symmetric matrix A, the matrix is divided into $nt = n/b$ block-columns of size nb. The algorithm proceeds panel by panel, performing a QR decomposition for each panel to generate the Householder reflectors V (i.e., the orthogonal transformations) required to zero out elements below the bandwidth nb. Then the generated block Householder reflectors are applied from the left and the right to the trailing symmetric matrix, according to

$$A = A - WV^T - VW^T, \tag{1.3}$$

where V and T define the block of Householder reflectors and W is computed as

$$\begin{aligned} W &= X - \tfrac{1}{2} VT^T V^T X, \text{where} \\ X &= AVT. \end{aligned} \tag{1.4}$$

Since the panel factorization consists of a QR factorization performed on a panel of size $l \times b$ shifted by nb rows below the diagonal, this will remove both the synchronization and the data dependency constraints seen using the classical one stage technique. In contrast to the classical approach, the panel factorization by itself does not require any operation on the data of the trailing matrix, making it an independent task. Moreover, we can factorize the next panel once we have finished its update, without waiting for the total trailing matrix update. Thus this kind of technique removes the bottlenecks of the classical approach: there are no BLAS-2 operations concerning the trailing matrix and also there is no need to wait for the update of the trailing matrix in order to start the next panel. However, the resulting matrix is banded, instead of tridiagonal. The hybrid CPU-GPU algorithm is illustrated in Fig. 1.14. We first run the QR decomposition (dgeqrf panel on step i of Fig. 1.14) of a panel on the CPUs. Once the panel factorization of step i is finished, then we compute W on the GPU, as defined by Eq. (1.4). In particular, it involves a dgemm to compute VT, then a dsymm to compute $X = AVT$, which is the dominant cost of computing W, consisting of 95 % of the time spent in computing W, and finally another inexpensive dgemm. Once W is computed, the trailing matrix update (applying transformations on the left and right) defined by Eq. (1.3) can be performed using a rank-$2k$ update.

However, to allow overlap of CPU and GPU computation, the trailing submatrix update is split into two pieces. First, the next panel for step $i+1$ (medium gray panel of Fig. 1.14) is updated using two dgemm's on the GPU. Next, the remainder of the trailing submatrix (dark gray triangle of Fig. 1.14) is updated using a dsyr2k. While the dsyr2k is executing, the CPUs receive the panel for step $i+1$, perform the next panel factorization (dgeqrf), and send the resulting V_{i+1} back to the GPU. In this way, the factorization of panels $i = 2, \ldots, nt$ and the associated communication are hidden by overlapping with GPU computation, as demonstrated in Fig. 1.15. This is similar to the look-ahead technique typically used in the one-sided dense

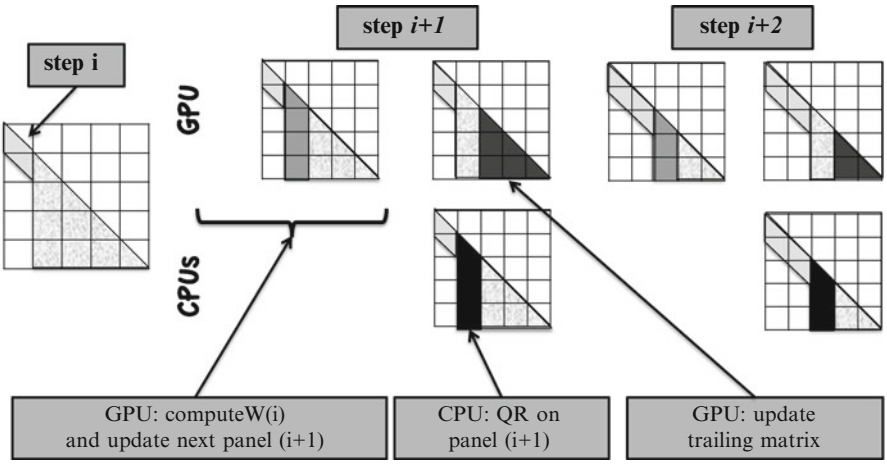

Fig. 1.14 Description of the reduction to band form, stage 1

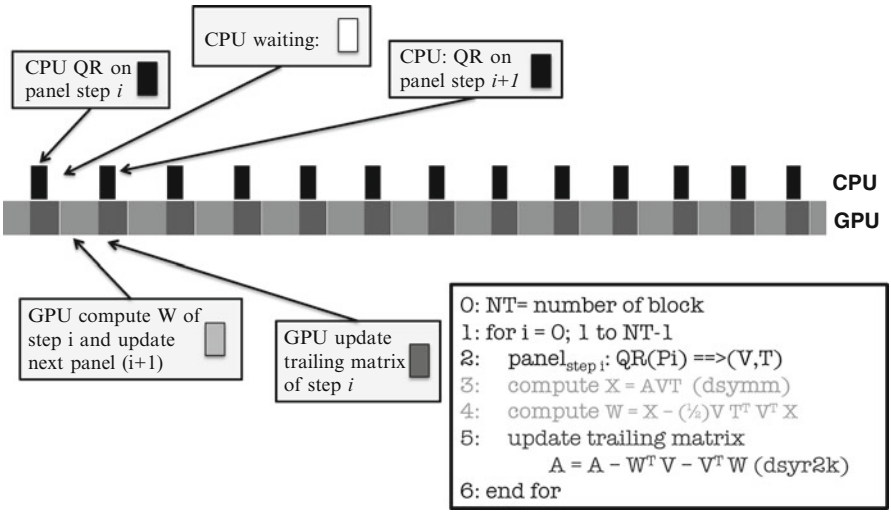

Fig. 1.15 Execution trace of reduction to band form

matrix factorizations. Figure 1.15 shows a snapshot of the execution trace of the reduction to band form, where we can easily identify the overlap between CPU and GPU computation. Note that the high-performance GPU is continuously busy, either computing W or updating the trailing matrix, while the lower performance CPUs wait for the GPU as necessary.

1.5.4.2 Second Stage: Cache-Friendly Computational Kernels

The band form is further reduced to the final condensed form using the bulge chasing technique. This procedure annihilates the extra off-diagonal elements by chasing the created fill-in elements down to the bottom right side of the matrix using successive orthogonal transformations. Each annihilation of the nb non-zero element below the off-diagonal of the band matrix is called a sweep. This stage involves memory-bound operations and requires the band matrix to be accessed from multiple disjoint locations. In other words, there is an accumulation of substantial latency overhead each time different portions of the matrix are loaded into cache memory, which is not compensated for by the low execution rate of the actual computations (the so-called surface-to-volume effect). To overcome these critical limitations, we developed a bulge chasing algorithm, to extensively use cache friendly kernels combined with fine grained, memory aware tasks in an out-of-order scheduling technique which considerably enhances data locality. This reduction has been designed for multicore architectures, and results have shown its efficiency. This step has been well optimized such that it takes between 5 and 10 % of the global time of the reduction from dense to tridiagonal. We refer the reader to [6, 8] for a detailed description of the technique.

We decide to develop a hybrid CPU-GPU implementation of only the first stage of the two stage algorithm, and leave the second stage executed entirely on the CPU. The main motivation is that the first stage is the most expensive computational phase of the reduction. Results show that 90 % of the time is spent in the first stage reduction. Another motivation for this direction is that accelerators perform poorly when dealing with memory-bound fine-grained computational tasks (such as bulge chasing), limiting the potential benefit of a GPU implementation of the second stage. Experiments showed that the two-stage algorithm can be up to six times faster than the standard one-stage approach.

1.5.5 Back Transform the Eigenvectors of the Two Stage Technique

The standard one-stage approach reduces the dense matrix A to condensed form (e.g., tridiagonal T in the case of symmetric matrix), computes its eigenvalues/eigenvectors (Λ, Z) and back transform its eigenvectors Z to computes the eigenvectors $X = Q\ Z$ of the original matrix A as mentioned earlier in Sect. 1.5.3. In the case of the two-stage approach, the first stage reduces the original dense matrix A to a band matrix by applying a two-sided transformations to A such that $Q_1^T A Q_1 = B$. Similarly, the second, bulge-chasing stage reduces the band matrix B to the condensed form (e.g, tridiagonal T) by applying two-sided transformations to B such that $Q_2^T B Q_2 = T$. Thus, when the eigenvectors matrix X of A are requested, the eigenvectors matrix Z resulting from the eigensolver needs to be back transformed by the Householder reflectors generated during the reduction phase, according to

$$X = Q_1 Q_2\ Z = (I - V_1 t_1 V_1^T)\ (I - V_2 t_2 V_2^T)\ Z, \tag{1.5}$$

where (V_1, t_1) and (V_2, t_2) represent the Householder reflectors generated during the reduction stages one and two, respectively. Note that when the eigenvectors are requested, the two stage approach has the extra cost of the back transformation of Q_2. However, experiments show that even with this extra cost the overall performance of the eigen/singular-solvers using the two stage approach can be several times faster than solvers using the one stage approach.

From the practical standpoint, the back transformation Q_2 is not as straightforward as the one of Q_1, which is similar to the classical back transformation described in Sect. 1.5.3. In particular, because of complications of the bulge-chasing mechanism, the order and the overlap of the Householder reflector generated during this stage is intricate. Let us first begin by describing the complexity and the design of the algorithm for applying Q_2. We present the structure of V_2 (the Householder reflectors that form the orthogonal matrix Q_2) in Fig. 1.16a. Note that

these reflectors represent the annihilation of the band matrix, and thus each is of length nb—the bandwidth size. A naïve implementation would take each reflector and apply it in isolation to the matrix Z. Such an implementation is memory-bound and relies on Level 2 BLAS operations. A better procedure is to apply with calls to Level 3 BLAS, which achieves both very good scalability and performance. The priority is to create compute intensive operations to take advantage of the efficiency of Level 3 BLAS. We proposed and implemented accumulation and combination of the Householder reflectors. This is not always easy, and to achieve this goal we must pay attention to the overlap between the data they access as well as the fact that their application must follow the specific dependency order of the bulge chasing procedure in which they have been created. To stress these issues, we will clarify it by giving an example. For sweep i (e.g., the column at position B(i,i):B(i+nb,i)), its annihilation generates a set of k Householder reflectors (v_i^k), each of length nb, the v_i^k are represented in column i of the matrix V_2 depicted in Fig. 1.16a. Likewise, the ones related to the annihilation of sweep $i + 1$, are those presented in column $i + 1$, where they are shifted one element down compared to those of sweep i. It is possible to combine the reflectors $v_i^{(k)}$ from sweep i with those from sweep $i + 1, i + 2, \ldots, i + \ell$ and to apply them together in blocked fashion. This grouping is represented by the diamond-shaped region in Fig. 1.16a. While each of those diamonds is considered as one block, their back transformation (application to the matrix Z) needs to follow the dependency order. For example, applying block 4 and block 5 of the V_2's in Fig. 1.16a modifies block row 4 and block row 5, respectively, of the eigenvector matrix Z drawn in Fig. 1.16b where one can easily observe the overlapped region. The order dictates that block 4 needs to be applied before block 5. It is possible to compute this phase efficiently by splitting Z by blocks of columns over both the CPUs and the GPU as shown in Fig. 1.16b, where we can apply each diamond independently to each portion of E. Moreover, this method does not require any data communication. The back transformation of Q_1 to the resulting matrix from above, $Q_1 \times (Q_2\, Z)$, involves efficient BLAS 3 kernels and it is done by using the GPU function magma_dormtr, which is the GPU implementation of the standard LAPACK function (dormtr).

1.6 Summary and Future Directions

In conclusion, GPUs can be used with astonishing success to accelerate fundamental linear algebra algorithms. We have demonstrated this on a range of algorithms, from the matrix–matrix multiplication kernel written in CUDA, to the higher level algorithms for solving linear systems, to eigenvalue and SVD problems. Further, despite the complexity of the hardware, acceleration was achieved at a surprisingly low software development effort using a high-level methodology of developing hybrid algorithms. The complete implementations and more are available through the MAGMA library. The promise shown so far motivates and opens opportunities for future research and extensions, e.g., tackling more complex algorithms and

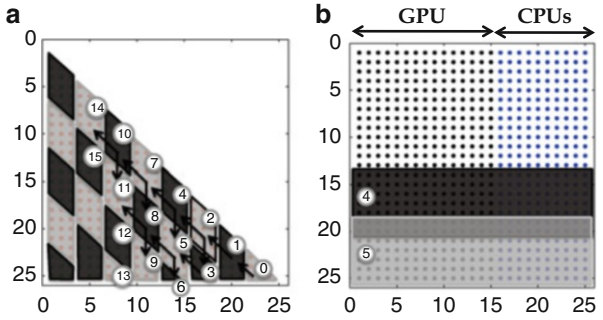

Fig. 1.16 Blocking technique to apply the Householder reflectors V_2 with a hybrid implementation on GPU and CPU. (**a**) Blocking for V_2; (**b**) eigenvectors matrix

hybrid hardware. Several major bottlenecks need to be alleviated to run at scale though, which is an intensive research topic. When a complex algorithm needs to be executed on a complex heterogeneous system, scheduling decisions have a dramatic impact on performance. Therefore, new scheduling strategies must be designed to fully benefit from the potential of future large-scale machines.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J.W., Dongarra, J.J. Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992). http://www.netlib.org/lapack/lug/
2. Bientinesi, P., Igual, F.D., Kressner, D., Quintana-Ortí, E.S.: Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In: Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I, PPAM'09, pp. 387–395. Springer, Berlin/Heidelberg (2010)
3. Dongarra, J.J., Sorensen, D.C., Hammarling, S.J.: Block reduction of matrices to condensed forms for eigenvalue computations. J. Comput. Appl. Math. **27**(1–2), 215–227 (1989)
4. Gansterer, W., Kvasnicka, D., Ueberhuber, C.: Multi-sweep algorithms for the symmetric eigenproblem. In: Vector and Parallel Processing - VECPAR'98. Lecture Notes in Computer Science, vol. 1573, pp. 20–28. Springer, Berlin (1999)
5. Golub, G., Loan, C.V.: Matrix Computations, 3rd edn. Johns Hopkins, Baltimore (1996)
6. Haidar, A., Ltaief, H., Dongarra, J.: Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In: Proceedings of SC '11, pp. 8:1–8:11. ACM, New York (2011)
7. Haidar, A., Ltaief, H., Luszczek, P., Dongarra, J.: A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, Shanghai, 21–25 May 2012. ISBN 978-1-4673-0975-2
8. Haidar, A., Tomov, S., Dongarra, J., Solca, R., Schulthess, T.: A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. Int. J. High Perform. Comput. Appl. **28**(2), 196–209 (2014)

9. Haidar, A., Kurzak, J., Luszczek, P.: An improved parallel singular value algorithm and its implementation for multicore hardware. In: SC13, The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, 17–22 November 2013
10. Lang, B.: Efficient eigenvalue and singular value computations on shared memory machines. Parallel Comput. **25**(7), 845–860 (1999)
11. Ltaief, H., Luszczek, P., Haidar, A., Dongarra, J.: Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) Proceedings of 9th International Conference, PPAM 2011, Torun, vol. 7203, pp. 661–670 (2012)
12. MAGMA 1.4.1: http://icl.cs.utk.edu/magma/ (2013)
13. Nath, R., Tomov, S., Dong, T., Dongarra, J.: Optimizing symmetric dense matrix-vector multiplication on GPUs. In: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–10. New York, NY, USAm 2011, ACM
14. Strazdins, P.E.: Lookahead and algorithmic blocking techniques compared for parallel matrix factorization. In: 10th International Conference on Parallel and Distributed Computing and Systems, IASTED, Las Vegas, 1998
15. Strazdins, P.E.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Int. J. Parallel Distrib. Syst. Netw. **4**(1), 26–35 (2001)
16. Tomov, S., Nath, R., Dongarra, J.: Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. Parallel Comput. **36**(12), 645–654 (2010)
17. Yamazaki, I., Dong, T., Solcà, R., Tomov, S., Dongarra, J., Schulthess, T.: Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. Concurr. Comput. Pract. Exp. (2013). doi:10.1002/cpe.3152