Chapter 4

BLAS for GPUs

Rajib Nath

Department of Electrical Engineering and Computer Science, University of Tennessee

Stanimire Tomov

Department of Electrical Engineering and Computer Science, University of Tennessee

Jack Dongarra

Department of Electrical Engineering and Computer Science, University of Tennessee
Computer Science and Mathematics Division, Oak Ridge National Laboratory
School of Mathematics & School of Computer Science, Manchester University

4.1 Introduction

Recent activities of major chip manufacturers, such as Intel, AMD, IBM and NVIDIA, make it more evident than ever that future designs of microprocessors and large HPC systems will be hybrid/heterogeneous in nature, relying on the integration (in varying proportions) of two major types of components:

1. Multi-/many-cores CPU technology, where the number of cores will con-

tinue to escalate while avoiding the power wall, instruction level parallelism wall, and the memory wall [1]; and

2. Special purpose hardware and accelerators, especially GPUs, which are in commodity production, have outpaced standard CPUs in performance, and have become as easy—if not easier—to program than multicore CPUs.

The relative balance between these component types in future designs is not clear, and will likely vary over time, but there seems to be no doubt that future generations of computer systems, ranging from laptops to supercomputers, will consist of a composition of heterogeneous components.

These hardware trends have inevitably brought up the need for updates on existing legacy software packages, such as the sequential LAPACK [2], from the area of dense linear algebra (DLA). To take advantage of the new computational environment, successors of LAPACK must incorporate algorithms of three main characteristics: high parallelism, reduced communication, and heterogeneity-awareness. In all cases though, the development can be streamlined if the new algorithms are designed at a high level (see Chapter 3), using just a few, highly optimized low-level kernels. Chapter 3 demonstrated a hybridization approach that indeed streamlined the development of high-performance DLA for multicores with GPU accelerators. The new algorithms, covering core DLA routines, are now part of the MAGMA library [3], a successor to LAPACK for the new heterogeneous/hybrid architectures. Similarly to LAPACK, MAGMA relies on the efficient implementation of a set of low-level linear algebra kernels. In the context of GPU-based hybrid computing, this is a subset of BLAS [4] for GPUs.

The goal of this chapter is to provide guidance on **how to develop high performance BLAS for GPUs**—a key prerequisite to enabling GPU-based hybrid approaches in the area of DLA. Section 4.2 describes some of the basic principles on how to write high-performance BLAS kernels for GPUs. Section 4.3 gives GPU-specific, generic kernel optimization techniques—pointer redirecting, padding, and auto-tuning—and their application in developing high-performance BLAS for GPUs. Finally, Section 4.4 gives a summary.

4.2 BLAS Kernels Development

Implementations of the BLAS interface are a major building block of DLA libraries, and therefore must be highly optimized. This is true for GPU computing as well, especially after the introduction of shared memory in modern GPUs. This is important because it enabled fast Level 3 BLAS implementations [5–7], which in turn made possible the development of DLA for GPUs to be based on BLAS for GPUs (see Chapter 3 and references [3, 6]).

Despite the current success in developing highly optimized BLAS for GPUs [5–7], the area is still new and presents numerous opportunities for improvements. Addressed are several very important kernels, including the matrix-matrix multiplication, crucial for the performance throughout DLA, and matrix-vector multiplication, crucial for the performance of linear solvers and two-sided matrix factorizations (and hence eigen-solvers). The new implementations are included in the MAGMA version 0.2 BLAS Library [3].

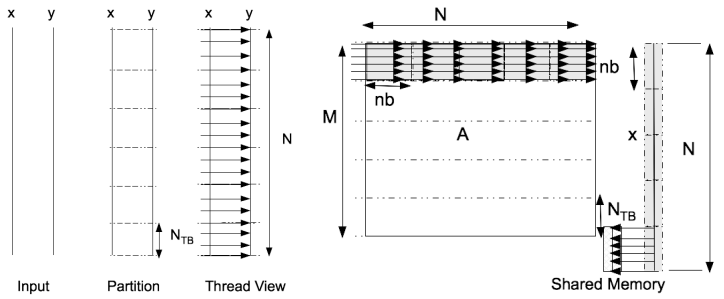
This section describes some of the basic principles on how to write high-performance kernels for GPUs. Along with the specifics on developing each of the BLAS considered, the stress is on two important issues for achieving high performance. Namely, these are:

Blocking Blocking is a DLA optimization technique where a computation is organized to operate on blocks/submatrices of the original matrix. The idea is that blocks are of small enough size to fit into a particular level of the CPU's memory hierarchy so that once loaded to reuse the blocks' data to perform the arithmetic operations that they are involved in. This idea can be applied for GPUs, using GPUs' shared memory. As demonstrated below, the application of blocking is crucial for the performance of numerous GPU kernels.

Coalesced Memory Access GPU global memory accesses are costly and not cached, making it crucial for the performance to have the right access pattern to get maximum memory bandwidth. There are two access requirements [8]. The first is to organize global memory accesses in terms of parallel consecutive memory accesses—16 consecutive elements at a time by the threads of a half-warp (16 threads)—so that memory accesses (to 16 elements at a time) are coalesced into a single memory access. This is demonstrated in the kernels' design throughout the section. Second, the data should be properly aligned. In particular, the data to be accessed by half-warp should be aligned at $16 * \texttt{sizeof(element)}$, e.g., 64 for single precision elements.

Clearly, fulfilling the above requirements will involve partitioning the computation into blocks of fixed sizes (e.g., multiple of 16) and designing memory accesses that are coalescent (properly aligned and multiple of 16 consecutive elements). This is demonstrated in the kernels' design throughout the section. The problems of selecting best performing partitioning sizes/parameters for the various algorithms as well as the cases where (1) the input data are not aligned to fulfill coalescent memory accesses and (2) the problem sizes are not divisible by the partitioning sizes required for achieving high performance need special treatment and are considered in Section 4.3. The main ideas in this section are demonstrated on general and symmetric matrices, in both the transpose and non-transpose cases.

The BLAS considered are not exhaustive; only subroutines that are critical for the performance of MAGMA are discussed. Moreover, these would often



(a) xAXPY (b) xGEMV (non-transpose)

FIGURE 4.1: Algorithmic view of Level 1 and Level 2 BLAS.

be DLA-specific cases that can be accelerated compared to CUBLAS [5], an implementation of the BLAS standard provided by NVIDIA.

Further down a *thread block* will be denoted by TB, its size by N_{TB} (or $N_{TBX} \times N_{TBY}$ in 2D), the number of threads in a TB by N_T (or $N_{TX} \times N_{TY}$ in 2D), and the size associated with blocking (as described above) by nb.

4.2.1 Level 1 BLAS

Implementing Level 1 BLAS, especially reduce-type operations like dot-product, isamax, etc., is of general interest for parallel computing, but not in the area of DLA. The reason is that Level 1 BLAS are of very low computational intensity (flops *vs* data required) and are avoided at first place (at algorithm design level) in DLA. Even when they cannot be avoided algorithmically, e.g., the use of isamax in LU for pivoting, their computation on the GPU is avoided by scheduling their execution on the CPU (see the hybrid approach described in Chapter 3). One operation that fits very well with the GPU architecture, and therefore can be efficiently executed on GPUs, is xAXPY:

$$y := \alpha x + y,$$

where x and y are vectors of size N, and α is a scalar. An example of its use is the mixed-precision iterative refinement solvers in MAGMA [9].

The implementation is straightforward—one dimensional TB of size N_{TB} computes N_{TB} consecutive elements of the resulting vector y (a thread per element; also illustrated in Figure 4.1(a)). Important for achieving high performance in this case, as discussed at the beggining of this section, is coalesced memory accesses, tuning N_{TB} and properly handling the case when N is not divisible by N_{TB} (i.e., $N \,\%\, N_{TB} \neq 0$). These are recurring issues for obtaining high-performance BLAS and will be further discussed in the context of other BLAS kernels and GPU optimization techniques like auto-tuning (in Section 4.3.3) and pointer redirecting (in Section 4.3.1).

Note that the algorithm described satisfies the first requirement for coalescent memory access—to organize global GPU memory accesses in terms of parallel consecutive memory accesses. The pointer redirecting technique in Section 4.3.1 deals with the second requirement for coalescent memory access, namely cases where the starting address of x is not a multiple of $16 * \texttt{sizeof(element)}$ and/or $N \% N_{TB} \neq 0$. The same applies for the other BLAS kernels in the section and will not be explicitly mentioned again.

4.2.2 Level 2 BLAS

Level 2 BLAS routines, similar to Level 1 BLAS, are of low computational intensity and, ideally, DLA algorithms must be designed to avoid them. An example from the area of DLA is the *delayed update* approach where the application of a sequence of Level 2 BLAS is delayed and accumulated in order to be applied at once as a more efficient single matrix-matrix multiplication [2]. In many cases, like MAGMA's mixed-precision iterative refinement solvers [9] or two-sided matrix factorizations [10], this is not possible, and efficient implementations are crutial for the performance. This section considers the GPU implementations of two fundamental Level 2 BLAS operations, namely the matrix-vector multiplication routines for correspondingly general (xGEMV) and symmetric matrices (xSYMV).

4.2.2.1 xGEMV

The xGEMV matrix-vector multiplication routine performs one of:

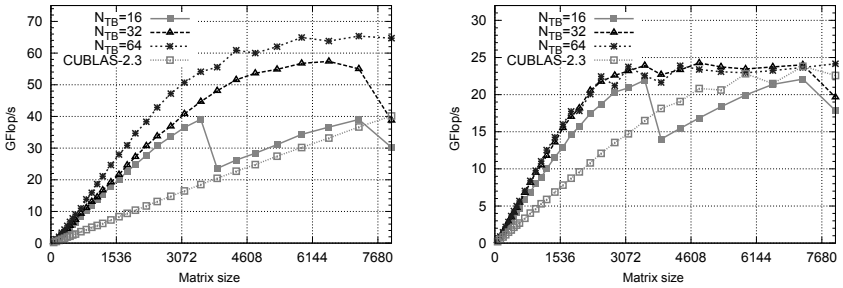
$$y := \alpha Ax + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y,$$

where A is an M by N matrix, x and y are vectors, and α and β are scalars. The two cases are considered separately as follows:

Non-Transposed Matrix: The computation in this case can be organized in one-dimensional grid of TBs of size N_{TB} where each block has $N_T = N_{TB}$ threads, as shown in Figure 4.1(b). Thus, each thread computes one element of the resulting vector y.

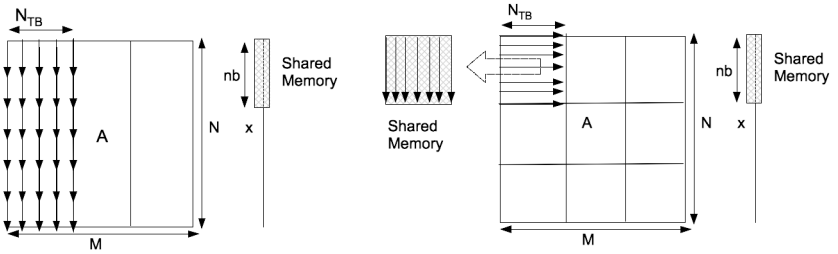
GEMV is the first of the kernels considered to which *blocking* can be applied. Although matrix A cannot be reused in any blocking, vector x can be reused by the threads in a TB. Specifically, the computation is blocked by loading nb consequtive elements of x at a time into the shared memory (using all N_T threads). This part of x is then used by all T_N threads in a TB to multiply it by the corresponging $N_{TB} \times nb$ submatrix of A. The process is repeated $\frac{N}{N_{TB}}$ times.

Note that the algorithm as described depends on two parameters: N_{TB} and nb. Figures 4.2(a), 4.2(b) compare the performance for cases $N_{TB} = nb = 16, 32, 64$ with that of CUBLAS 2.3. The performances are for matrix sizes $M = N$ that are divisible by the corresponding blocking sizes. Also, the starting addresses of A, x, and y are taken to be divisible by $16 * \texttt{sizeof(element)}$



(a) Single Precision (b) Double Precision

FIGURE 4.2: Performance of xGEMV (non-transpose) on a GTX 280.



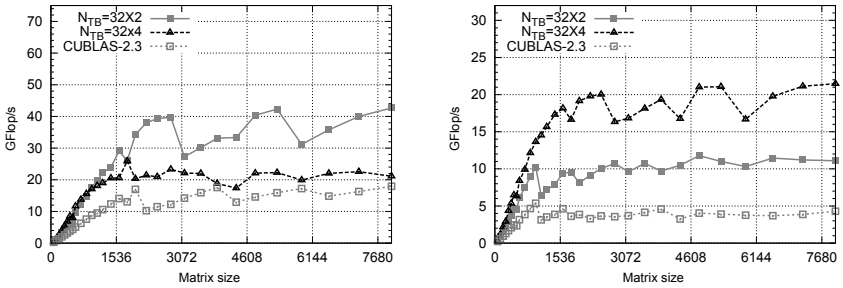
(a) Basic implementation (b) Optimized implementation

FIGURE 4.3: Two memory access implementations of xGEMV (transpose).

and the leading dimension of A is divisible by 16. This guarantees that all memory accesses in the algorithm are coalescent.

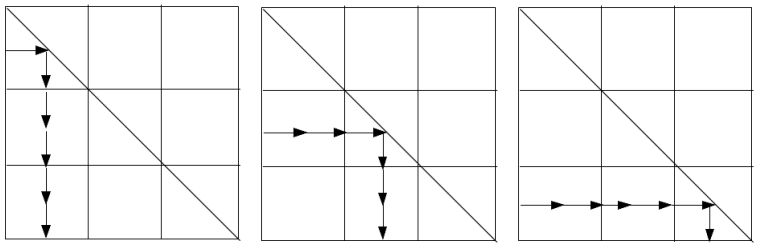
Transposed Matrix: Following the non-transposed version approach leads to poor performance because the memory acceses are not going to be coalesced (see Figure 4.3(a)). To improve the speed on accessing the data, blocks of the matrix A can be first loaded into the shared memory using coalesced memory accesses, and second, data only from the shared memory can be used to do all the necessary computations (see Figure 4.3(b)).

Although the new version significantly improves the performance, experiments that increase the design space of the algorithm show that further improvements are possible. In particular, one exploration direction is the use of higher numbers of threads in a TB, e.g., 64, as high-performance DLA kernels are associated with the use of 64 threads (and occasionally more). Using 64 threads directly does not improve performance though because the amount of shared memory used (a 64×64 matrix) gets to be excessive, prohibiting the effective scheduling of that amount of threads [8]. Decreasing the use of shared memory, e.g., to a 32×32 matrix, while having a higher level of thread parallelism, e.g., a grid of 32×2 threads, is possible in the following way:



(a) Single Precision (b) Double Precision

FIGURE 4.4: Performance of xGEMV (transpose) on a GTX 280.



(a) TYPE A (b) TYPE B (c) TYPE C

FIGURE 4.5: Three cases of TB computations in xSYMV.

(1) two groups of 32×1 threads, e.g., denoted by 32_j where $j = 0/1$, load correspondingly the two 32×16 submatrices of the shared memory matrix using coalesced memory accesses, (2) each group performs the computation from the second GEMV version but constrained to the 16×32 submatrix of the shared memory matrix, accumulating their independent y_j results. The final result $y := y_0 + y_1$ can be accumulated by one of the $j = 0/1$ threads.

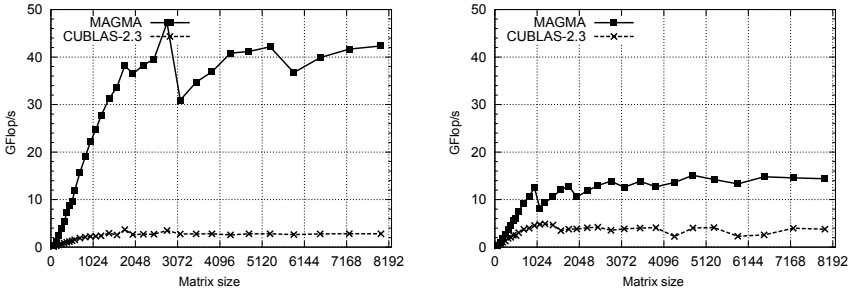
The same idea can be used with more threads, e.g., 32×4, while using the same amount of shared memory. Performance results are shown in Figure 4.4 along with a comparison to the performance from CUBLAS 2.3.

4.2.2.2 xSYMV

The xSYMV matrix-vector multiplication routine performs:

$$y := \alpha Ax + \beta y,$$

where α and β are scalars, x and y are vectors of size N, and A is an N by N symmetric matrix, stored in the upper or lower triangular part of a two-dimensional array of size $N \times N$. The difficulty of designing a high-performance SYMV kernel stems from the triangular data storage, which is more challenging to organize a data parallel computation with coalescent memory accesses.



(a) Single Precision (b) Double Precision

FIGURE 4.6: Performance of xSYMV on a GTX 280.

Indeed, if A is given as an $N \times N$ array, storing both the upper and lower triangular parts of the symmetric matrix A, the SYMV kernel can be implemented using GEMV. Similar to GEMV, the computation is organized in one-dimensional grid of TBs of size N_{TB}, where each block has $N_T = N_{TB}$ threads. A TB computation can be classified as one of three cases (see the illustration in Figure 4.5):

- Type A—TB threads do SYMV followed by GEMV (transpose);
- Type B—threads do GEMV (non-transpose) followed by SYMV and GEMV (transpose);
- Type C—threads do GEMV (non-transpose) followed by SYMV.

This way the computation within a TB is converted into one/two GEMVs (to reuse the GEMV kernels) and an SYMV involving a matrix of size $N_{TB} \times N_{TB}$. The remaining SYMV is also converted into a GEMV by loading the $N_{TB} \times N_{TB}$ matrix into the GPU's shared memory and generating the missing symmetric part in the shared memory (a process defined as *mirroring*). Figure 4.6 compares the performance for kernel with parameters $N_{TB} = nb = 32$, $N_T = 32 \times 4$ with that of CUBLAS 2.3.

4.2.3 Level 3 BLAS

Level 3 BLAS routines are of high computational intensity, enabling their implementations (and that of high-level DLA algorithms based on Level 3 BLAS) to get close within the computational peak of ever evolving architectures, despite that architectures are evolving with an exponentially growing gap between their compute and communication speeds. The shared memory of GPUs, similar to memory hierarchy in standard CPUs, can be used to develop highly efficient Level 3 BLAS kernels. This section describes the GPU implementations of three primary Level 3 BLAS operations: the matrix-matrix

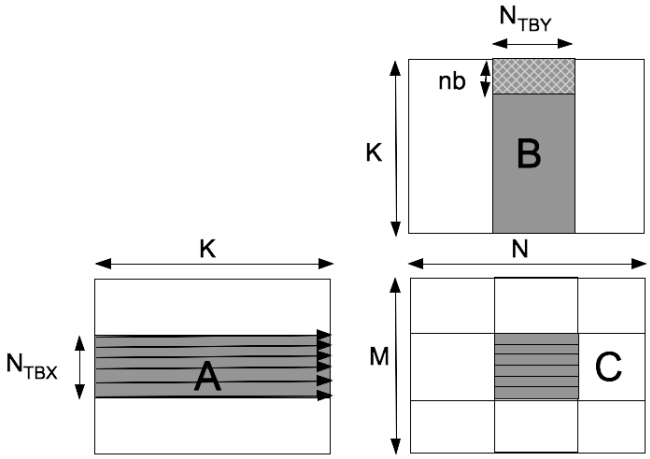


FIGURE 4.7: The GPU GEMM ($C = AB$) of a single TB.

multiplication (xGEMM), the symmetric rank-k update (xSYRK), and the triangular matrix solver (xTRSM).

4.2.3.1 xGEMM

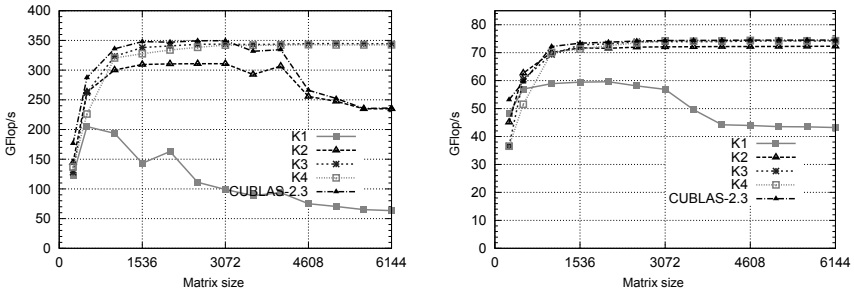
The xGEMM matrix-matrix multiplication routine performs one of:

$$C := \alpha\ op(A)op(B) + \beta C,$$

where $op(X)$ is X, or X^T, α and β are scalars; and A, B, and C are matrices. Crutial for the performance is the application of blocking—schematicly represented in Figure 4.7 for the case of $C := \alpha AB + \beta C$ and described as follows [6]. The computation is done on a two-dimensional grid of TBs of size $N_{TBX} \times N_{TBY}$ and each TB is assigned to $N_T = N_{TX} \times N_{TY}$ threads. For simplicity, take $N_T = N_{TBX}$. Then, each thread is coded to compute a row of the submatrix assigned to the TB. Each thread accesses its corresponding row of A, as shown by an arrow, and uses the $K \times N_{TBY}$ submatrix of B for computing the final result. This TB computation can be blocked, which is crucial for obtaining high performance. In particular, submatrices of B of size $nb \times N_{TBY}$ are loaded into shared memory and multiplied nb times by the corresponding $N_{TBX} \times 1$ submatrices of A. The $N_{TBX} \times 1$ elements are loaded and kept in registers while multiplying them with the $nb \times N_{TBY}$ part of B. The result is accumulated to the resulting $N_{TBX} \times N_{TBY}$ submatrix of C, which is kept in registers throughout the TB computation (a row per thread, as already mentioned). This process is repeated until the computation is over. All memory accesses are coalesced. Kernels for various $N_{TBX}, N_{TBY}, N_{TX}, N_{TY}$, and nb can be automatically generated (see Section 4.3.3) to select the best performing kernel for particular architecture and GEMM parameters. A sam-

Kernel	N_{TBX}	N_{TBY}	nb	N_{TX}	N_{TY}
K1	32	8	4	8	4
K2	64	16	4	16	4
K3	128	16	8	16	8
K4	256	16	16	16	16

TABLE 4.1: Key parameters of a sample of GPU GEMM kernels.



(a) Single Precision (b) Double Precision

FIGURE 4.8: Performance of GEMM ($C = \alpha AB^T + \beta C$) on a GTX 280.

ple choice of these kernels is shown in Table 4.1. Figure 4.8 compares their performances with that of CUBLAS 2.3 on square matrices. K1 performs well for small matrices (e.g., of dimension ≤ 512) as it provides more parallelism compared to the other kernels in Table 4.1. The performance deteriorations experienced by some of the kernels are due to the GPUs global memory layout and memory access patterns of hitting a particular memory module (a phenomenon referred to by NVIDIA as *partition camping*).

This particular configuration works well when $Op(A) = A,\ Op(B) = B$. The $Op(A) = A^T,\ Op(B) = B^T$ case is similar—only the argument order and the update location of C at the end of the kernel have to be changed, as:

$$C := \alpha\ A^T B^T + \beta C \quad \text{or} \quad C^T := \alpha\ BA + \beta C^T.$$

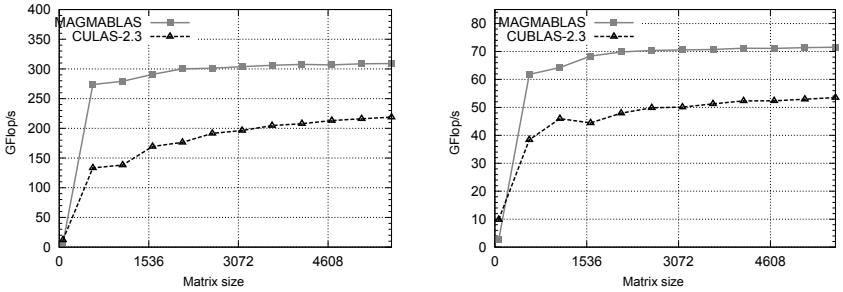
The $Op(A) = A^T,\ Op(B) = B$ kernel can be analogously developed except that both A and B must be stored into shared memory.

4.2.3.2 xSYRK

The xSRYK routine performs one of the symmetric rank-k updates:

$$C := \alpha AA^T + \beta C \quad \text{or} \quad C := \alpha A^T A + \beta C,$$

where α and β are scalars, C is an $N \times N$ symmetric matrix, and A is an $N \times K$ matrix in the first case and a $K \times N$ matrix in the second case. A TB



(a) Single Precision (b) Double Precision

FIGURE 4.9: Performance of xSYRK on a GTX 280.

index reordering technique can be used to initiate and limit the computation only to TBs that are on the diagonal or in the lower (correspondingly upper) triangular part of the matrix. In addition, all the threads in a diagonal TB compute redundantly half of the block in a data parallel fashion in order to avoid expensive conditional statements that would have been necessary otherwise. Some threads also load unnecessary data to ensure coalescent global memory accesses. At the end, the results from the redundant computations (in the diagonal TBs) are discarded and the data tile is correctly updated. Figure 4.9 illustrates the performance gains in applying this technique.

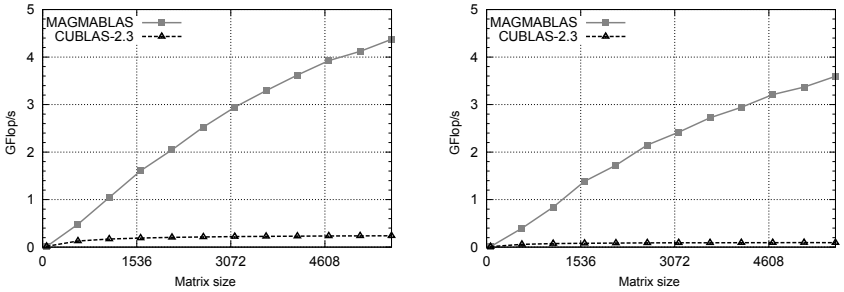
4.2.3.3 xTRSM

The xTRSM routine solves one of the matrix equations:

$$op(A)X = \alpha B \quad \text{or} \quad Xop(A) = \alpha B,$$

where α is a scalar, X and B are $M \times N$ matrices, A is upper/lower triangular matrix and $op(A)$ is A or A^T. Matrix B is overwritten by X.

Trading off parallelism and numerical stability, especially in algorithms related to triangular solvers, has been known and studied before [11,12]. Some of these TRSM algorithms are getting extremely relevant with the emerging highly parallel architectures, especially GPUs. In particular, the MAGMA library includes implementations that explicitly invert blocks of size 32×32 on the diagonal of the matrix and use them in blocked xTRSM algorithms. The inverses are computed simultaneously, using one GPU kernel, so that the critical path of the blocked xTRSM can be greatly reduced by doing it in parallel (as a matrix-matrix multiplication). Variations are possible, e.g., the inverses to be computed on the CPU, to use various block sizes, including recursively increasing it from 32, etc. Similarly to xSYRK, extra flops can be performed to reach better performance—the empty halves of the diagonal triangular matrices can be set to zeros and the multiplications with them done with GEMMs instead of with TRMMs. This avoids diverting warp threads and



(a) Single Precision (b) Double Precision

FIGURE 4.10: Performance of xTRSM on a GTX 280.

ensures efficient parallel execution. Figure 4.10 illustrates the performance gains in applying this technique.

4.3 Generic Kernel Optimizations

This section addresses three optimization techniques that are crutial for developing high performance GPU kernels. The first two techniques—pointer redirecting (Section 4.3.1) and padding (Section 4.3.2)—are used in cases where the input data are not aligned to directly allow coalescent memory accesses, or when the problem sizes are not divisible by the partitioning sizes required for achieving high performance. The third technique—auto-tuning (Section 4.3.3)—is used to determine best performing kernels, partitioning sizes, and other parameters for the various algorithms described.

An example demonstrating the need to address the cases mentioned is given in Figure 4.11. Shown is the performance of the matrix-matrix multiplication routine for a discrete set of matrix dimensions. Seen are performance deteriorations, e.g., more than 24 GFlops/s in double precision (around 30% of the peak performance) and even worse in single precision. The techniques in this section, used as a complement to the kernels presented in Section 4.2, aim to streamline the development of BLAS kernels for GPUs that are of uniformly high performance.

4.3.1 Pointer Redirecting

A few possibilities of dealing with matrix dimensions not divisible by the blocking factor can be explored. One approach is to have some "boundary" TBs doing selective computation. This will introduce several if-else statements

in the kernel which will prevent the threads inside a TB to run in parallel. Figure 4.12 shows the GEMM performance following this approach.

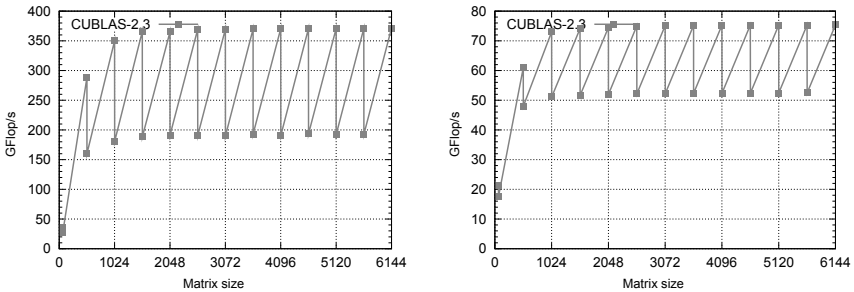
Another approach is instead of preventing certain threads from computing (with if-else statements), to let them do similar work as the other threads in a TB, and discard saving their results at the end. This can lead to some illegal memory references as illustrated in Figure 4.13.

The *pointer redirecting* techniques are based on the last approach and include redirecting the illegal memory references to valid ones, within the matrix of interest, in a way that would allow the memory accesses to be coalescent. The specifics of the redirecting depend on the kernel, but in general, if a thread is to access invalid rows/columns of a matrix (beyond row/column M/N), the access is redirected towards the last row/column.

Figure 4.14(a) shows the pointer redirecting for matrix A in GEMM with $Op(A) = A$ and $Op(B) = B$. Threads t1, t2, t3, and t4 access valid memory location, and threads beyond that, e.g., t5, t6 access the memory accessed by t4. As a result no separate memory read operation will be issued and no latency will be experienced for this extra load. Figure 4.14(b) shows the data access pattern for matrix B – $nb \times N_{TBY}$ data of matrix B is loaded into shared memory by $N_{TX} \times N_{TY}$ threads in a coalesced manner. The left $nb \times N_{TBY}$ block is needed, but the right $nb \times N_{TBY}$ is only partially needed. As discussed before, the illegal memory accesses will be redirected to the last column of B. The redirection as done presents a simple solution that has little overhead and does not break the pattern of coalesced memory access.

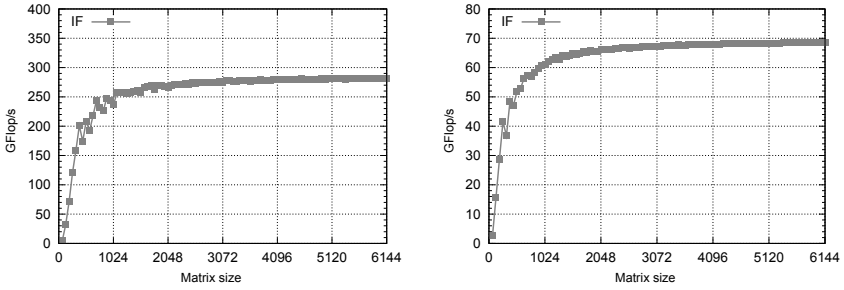
Figures 4.15 and 4.16 show the performance results for GEMM using the pointer redirecting technique. In double precision the performance is improved by up to 24 GFlops/s and in single precision by up to 170 GFlops/s.

The technique is applied similarly to the other kernels. The case where the starting address of any of the operands is not a multiple of $16 * sizeof(element)$ (but the leading dimension is a multiple of 16) is also handled similarly—threads that must access rows "before" the first are redirected to access the first.



(a) Single Precision (b) Double Precision

FIGURE 4.11: Performance of GEMM on square matrices on a GTX 280.



(a) Single Precision (b) Double Precision

FIGURE 4.12: Performance of GEMM with conditional statements.

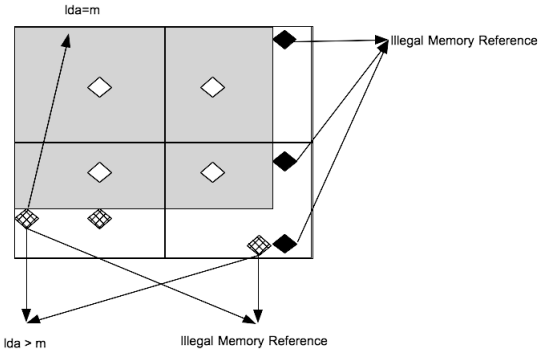
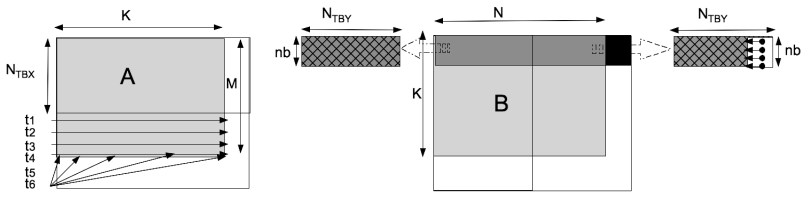
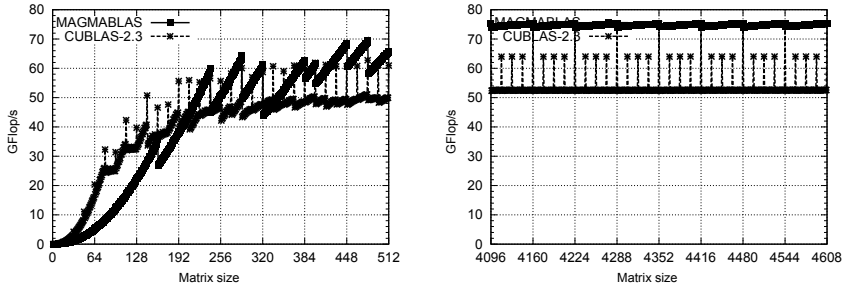


FIGURE 4.13: Possible illegal memory references in GEMM.



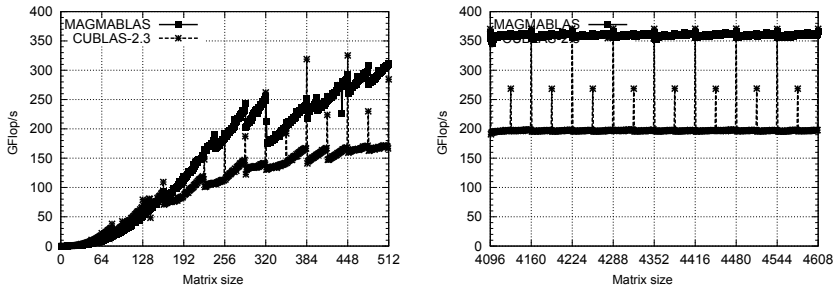
(a) Acessing matrix A (b) Accessing matrix B

FIGURE 4.14: GPU GEMM ($C = \alpha AB + \beta C$) with pointer redirecting.



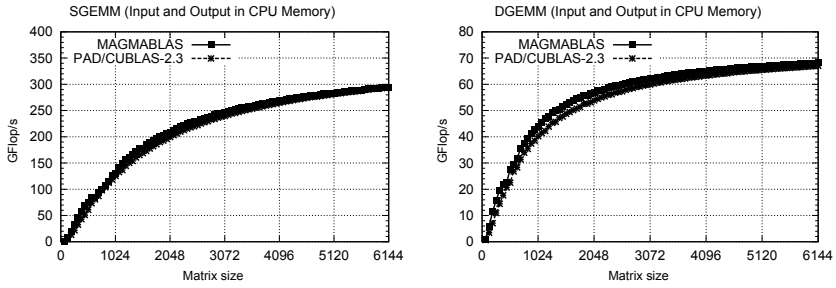
(a) Small Dimension (b) Large Dimension

FIGURE 4.15: Performance of DGEMM on a GTX 280.



(a) Small Dimension (b) Large Dimension

FIGURE 4.16: Performance of SGEMM on a GTX 280.



(a) SGEMM (b) DGEMM

FIGURE 4.17: Performance of xGEMM with padding on a GTX 280.

4.3.2 Padding

If the input matrices are given on the CPU memory the performance can be enhanced by *padding*, given that enough memory is available for it on the GPU. Padding is the technique where a matrix of higher dimension (to make the new size divisible by N_{TB}) is allocated on the GPU memory, and the extra elements initialized by zero. Figure 4.17 shows the performance of xGEMM comparing the padding and pointer redirecting approaches when the data are in the CPU memory. The results show that for small matrix sizes the pointer redirecting gives better performance, and for larger matrices the two approaches are almost identical, as it is actually expected.

Padding is applied to certain CPU-interface MAGMA routines [3], e.g., the LU factorization. In general, users of the CPU interface are encouraged to apply "padding" to all routines, in the sense that users must provide at least working space matrices on the GPU with leading dimensions divisible by 16. Note that if the leading dimension is not divisible by 16 none of the techniques presented will have coalescent memory accesses, unless internally the data are copied into another padded matrix. This second form of padding does not require zeroing the extra space—just allocating it so that coalescent memory accesses are enabled.

4.3.3 Auto-Tuning

Automatic performance tuning (optimization), or auto-tuning in short, is a technique that has been used intensively on CPUs to automatically generate near-optimal numerical libraries. For example, ATLAS [13, 14] and PHiPAC [15] are used to generate highly optimized BLAS. In addition, FFTW [16] is successfully used to generate optimized libraries for FFT, which is one of the most important techniques for digital signal processing. With the success of auto-tuning techniques on generating highly optimized DLA kernels on CPUs, it is interesting to see how the idea can be used to generate near-optimal DLA kernels on modern high-performance GPUs.

Indeed, work in the area [17] has already shown that auto-tuning for GPUs is a very practical solution to easily port existing algorithmic solutions on quickly evolving GPU architectures and to substantially speed up even highly hand-tuned kernels.

There are two core components in a complete auto-tuning system:

Code generator The code generator produces code variants according to a set of pre-defined, parametrized templates/algorithms. The code generator also applies certain state-of-the-art optimization techniques.

Heuristic search engine The heuristic search engine runs the variants produced by the code generator and finds out the best one using a feedback loop, e.g., the performance results of previously evaluated variants are used as a guideline for the search on currently unevaluated variants.

Below is a review of certain techniques and choices of parameters that significantly impact the performance of the GEMM kernel. Therefore, these techniques and parameters must be (and have been) incorporated into the code generator of an auto-tuning GEMM system. The ultimate goal is to develop similar auto-tuning for all of the BLAS of interest.

Auto-tuning GEMM: Figure 4.7 depicts the algorithmic view of a GEMM code template. It was already mentioned that five parameters can critically impact performance (see Table 4.1 for a sample choice), and therefore are incorporated in a GEMM code generator. This choice though can be extended and enhanced with various optimization techniques:

Number of threads computing a row: Section 4.2.3.1 imposed the constraint $N_{TX} \times N_{TY} = N_{TBX}$ so that each thread in a TB is computing an entire row of the submatrix of C computed by the TB (denoted further as BC). This constraint can be lifted to introduce an additional template parameter. Depending upon the value of N_T each thread will compute either an entire row or part of a row. For example, suppose $N_{TBY} = 16$ and $N_{TBX} = 64$, and the TB has 16×4 threads, then each thread will compute exactly one row of BC. If the thread block has 16×8 threads, then each thread will compute half of a row.

A/B being in shared memory: As described in Section 4.2.3.1, whether A or B is put into shared memory plays a crucial factor in the kernel's performance. Different versions of GEMM (`Op(X)` is `X` or $\texttt{X}^\texttt{T}$) require putting A and/or B into shared memory. This parameter of the auto-tuner is denoted by sh_{AB}. When only (part of) A is in shared memory each thread per TB computes an entire column or part of a column of BC. When both A and B are in shared memory the computation can be splitted in terms of rows or columns of the resulting submatrix of C.

Submatrix layout in shared memory: This parameter determines the layout of each $N_{TBX} \times nb$ submatrix of the matrix A (referred to as

BA from now on) or $N_{TBY} \times nb$ submatrix of the matrix B (referred to as BB from now on) in the shared memory, i.e., whether the copy of each block BA or BB in the shared memory is transposed or not. Since the shared memory is divided into banks and two or more simultaneous accesses to the same bank cause bank conflicts, transposing the layout in the shared memory may help reduce the possibility of bank conflicts, thus potentially improving the performance.

Amount of allocated shared memory: Two parameters, $offset_{BA}$ and $offset_{BB}$, relate to the actual allocation size of BA or BB in shared memory. When $N_{TBY} = 16$ and $nb = 16$, it requires 16×16 2D-array for BB in shared memory. Depending upon the computation sometimes it is better to allocate some extra memory so that the threads avoid bank conflict while accessing operands from shared memory data. It means allocating a 16×17 array instead of 16×16. So there is an offset of 1. It could be 0, 2, or 3 depending upon other parameters and the nature of computation. The auto-tuner handles this offset as a tunable parameter in internal optimization.

Prefetching into registers: As in CPU kernels, GPU kernels can benefit by prefetching into registers. For the access of matrices A and B, the auto-tuner inserts prefetch instruction for the data needed in the next iteration and checks the effect. Insertion of prefetch instruction leads to usage of registers which might limit the parallelism of the whole code. The auto-tuner investigates this with various combinations of prefetches: no prefetch, prefetch A only, prefetch B only, and prefetch both A and B, to finally pick the best combination.

Loop optimization techniques: Different state-of-the-art loop optimization techniques such as strip mining and loop unrolling are incorporated in order to extract parallelism and achieve performance. Another interesting loop optimization technique, namely *circular loop skewing*, was incorporated in the auto-tuner to deal with GPU global memory layout. Circular loop skewing is based upon a very simple idea of reordering the computation in the inner loop. In the context of GPUs, inner loops are considered the data parallel tasks that make up a kernel. These tasks are scheduled by CUDA (controlling the outer loop) on the available multiprocessors and the order of scheduling sometimes is crucial for the performance. Circular loop skewing techniques are incorporated to explore benefits of modified scheduling. Their most important use is in removing performance deteriorations related to *partition camping* (described above).

Precision: The code generator also takes precision as a parameter.

The code generator takes all these parameters as input and generates the

Kls	Prec	N_{tbx}	N_{tby}	nb	N_{tx}	N_{ty}	sh_{AB}	$Trns$	$op(A)$	$op(B)$	$skewing$
K1	S/DP	32	8	4	8	4	B	No	N	T	No
K2	S/DP	64	16	4	16	4	B	No	N	T	No
K3	S/DP	128	16	8	16	8	B	No	N	T	No
K4	S/DP	256	16	16	16	16	B	No	N	T	No
K5	DP	32	32	8	8	8	AB	No	T	N	No
K6	DP	64	16	16	16	4	B	Yes	N	N	No
K7	DP	128	16	8	16	8	B	Yes	N	N	No
K8	SP	64	16	4	16	4	B	No	N	T	All
K9	SP	64	16	4	16	4	B	No	N	T	Selective

TABLE 4.2: Different kernel configurations.

kernel, the timing utilities, the header file, and the Makefile to build the kernel. The code generator first checks the validity of the input parameters before actually generating the files. By validity it means (1) the input parameters conform to hardware constraints, e.g., the maximum number of threads per thread block $N_{TX} \times N_{TY} \leq 512$ in GTX 280, and (2) the input parameters are mutually compatible, e.g., $(N_{TBX} \times N_{TBY})\%(N_{TX} \times N_{TY}) = 0$, i.e., the load of BA's data into share memory can be evenly distributed among all the threads in a thread block, etc. By varying the input parameters, the auto-tuner can generate different versions of the kernel, and evaluate their performance, in order to identify the best one. Along the way the auto-tuner tries to optimize the code by using different optimization techniques such as prefetching, circular loop skewing and adjusting offset in shared memory allocation as described above. One way to implement auto-tuning is to generate a small number of variants for some matrices with typical sizes during installation time, and choose the best variant during run time, depending on the input matrix size and high-level DLA algorithm.

Performance results: Table 4.2 gives the parameters of different xGEMM kernels used in this section. The table also provides parameters for all the kernels used in Section 4.2.3.1. The *Trns* parameter denotes if the kernel was implemented by taking tranpose operation in both sides of the equation of the original operation, as:

$$C := \alpha\ A^T B^T + \beta C \quad \text{or} \quad C^T := \alpha\ BA + \beta C^T.$$

Figure 4.18 compares the performance of the xGEMM auto-tuner in double precision with the CUBLAS 2.3 for multiplying square matrices where $Op(A) = A^T$ and $Op(B) = B$. It can be seen that the performance of the auto-tuner is apparently 15% better than the CUBLAS 2.3 DGEMM. The fact that the two performances are so close is not surprising because the auto-tuned code and CUBLAS 2.3's code are based on the same kernel, and this kernel was designed and tuned for current GPUs (and in particular the GTX 280), targeting high performance for large matrices.

The global memory layout of current GPUs presents challenges as well as opportunities for auto-tuners. As shown in Figure 4.19(a), CUBLAS 2.3 SGEMM has performance deteriorations for certain problem sizes when

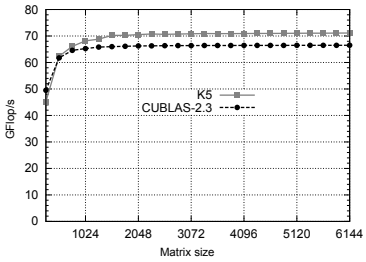
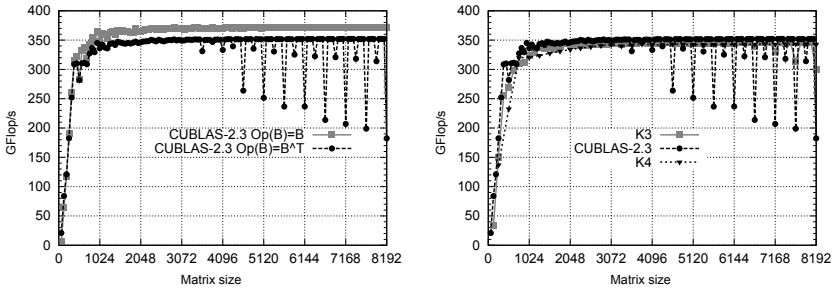
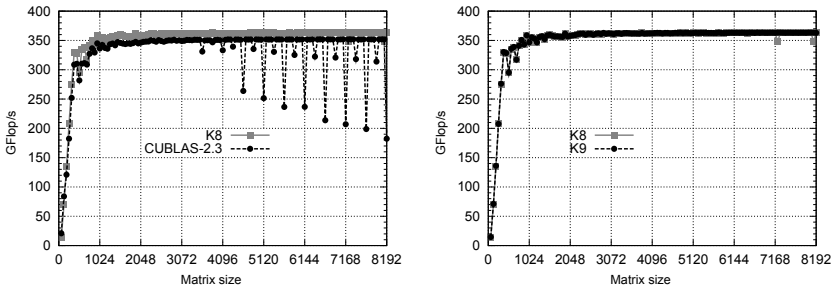


FIGURE 4.18: Performance of auto-tuned DGEMM kernel ($Op(A) = A^T$, $Op(B) = B$) on a GTX 280.



(a) Performance comparison of SGEMM kernel between $Op(B) = B$ and $Op(B) = B^T$ with $Op(A) = A$.

(b) Auto-tuned kernel with tuned algorithmic parameter.



(c) Auto-tuned kernel with circular skewing in all dimensions.

(d) Auto-tuned kernel with selective circular skewing.

FIGURE 4.19: Performance of the auto-tuned SGEMM ($Op(A) = A$, $Op(B) = B^T$) kernel for square matrices on a GTX 280.

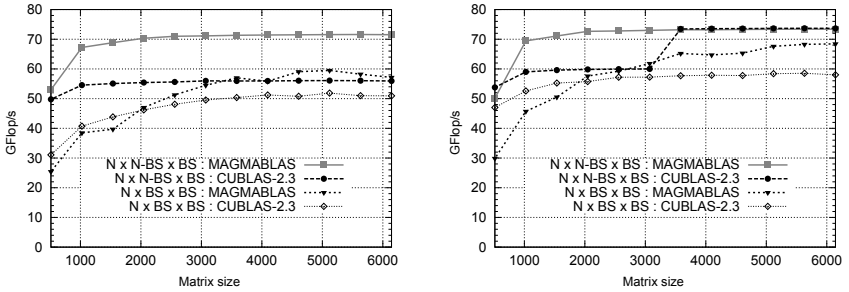
$Op(A) = A$ and $Op(B) = B^T$. Interestingly, when $Op(A) = A$ and $Op(B) = B$, the performance is very smooth. The reason for this is that GPU global memory is interleaved into a number of memory modules and the memory requests from all the concurrently running thread blocks may not be evenly distributed among the GPU memory modules. As a result the memory requests are sequentially processed and all the threads experience huge memory latency. This phenomenon is referred to as *partition camping* in NVIDIA terms. The auto-tuner found two kernels ($K3$, $K4$), as shown in Figure 4.19(b), that work significantly better in this situation. $K3$ and $K4$ work better because as partition size N_{TBX} is increased, the total number of accesses to global memory for matrix B's data is correspondingly 1/2 and 1/4 compared to that for kernel $K2$ (besides, TLP is increased). Kernels $K3$ and $K4$ perform fair compared to CUBLAS 2.3 in any dimension, and remarkably well for the problem sizes where CUBLAS 2.3 has performance deteriorations. Interestingly, the auto-tuner was successful in finding a better kernel by applying circular loop skew optimization in kernel $K2$. The performance is shown in Figure 4.19(c). Note that there are no performance deteriorations and performance is better than CUBLAS 2.3 for all matrix sizes. However, this technique does not work in all cases and may have to be applied selectively. The performance of such kernel (K9) is shown in Figure 4.19(d).

Finally, we point out that in the area of DLA, it is very important to have high-performance GEMMs on rectangular matrices, where one size is large- and the other is fixed within a certain block size (BS), e.g., BS = 64, 128, up to about 256 on current architectures. For example, in an LU factorization (with look-ahead) it requires two types of GEMM, namely one for multiplying matrices of size N×BS and BS×N−BS, and another for multiplying N×BS and BS×BS matrices. This situation is illustrated on Figure 4.20, where we compare the performances of the CUBLAS 2.3 *vs* auto-tuned DGEMMs occurring in the block LU factorization of a matrix of size 6144×6144. The graphs show that the auto-tuned code significantly outperforms (up to 27%) the DGEMM from CUBLAS 2.0.

These results support experiences and observations by others on "how sensitive the performance of GPU is to the formulations of your kernel" [18] and that an enormous amount of well-thought experimentation and benchmarking [6, 18] is needed in order to optimize performance.

4.4 Summary

Implementations of the BLAS interface are a major building block of dense linear algebra libraries, and therefore must be highly optimized. This is true for GPU computing as well, as evident from the MAGMA library, where the availability of fast GPU BLAS enabled a hybridization approach that stream-



(a) BS=64 (b) BS=128

FIGURE 4.20: Performance comparison of the auto-tuned (solid line) *vs* CUBLAS 2.3 DGEMMs occurring in the block LU factorization (for block sizes BS = 64 on the left and 128 on the right) of a matrix of size 6144×6144. The two kernels shown are for multiplying N×BS and BS×N−BS matrices (denoted by N×N−BS×BS), and N×BS and BS×BS matrices (denoted by N×BS×BS). K6 was used when BS = 64 and K7 was used when BS = 128.

lined the development. This chapter provided guidance on how to develop these needed high-performance BLAS kernels for GPUs. Described were not only basic principles and important issues for achieving high performance, but also specifics on the development of each of the BLAS considered. In particular, the stress was on the two important issues—blocking and coalesced memory access—demonstrated in the kernels' design throughout the chapter. The main ideas were demonstrated on general and symmetric matrices, in both the transpose and non-transpose cases, for a selection of Level 1, 2, and 3 BLAS kernels that are crucial for the performance of higher-level DLA algorithms. Moreover, three optimization techniques that are GPU specific and crucial for developing high performance GPU kernels, were considered. The first two techniques, pointer redirecting and padding, are used in cases where the input data is not aligned to directly allow coalescent memory accesses, or when the problem sizes are not divisible by the partitioning sizes required for achieving high performance. The third technique, auto-tuning, is used to automate the process of generating and determining best performing kernels, partitioning sizes, and other parameters for the various algorithms of interest.

The implementations of variation of the kernels described are available as part of MAGMA BLAS library through the MAGMA site http://icl.eecs.utk.edu/magma/.

Bibliography

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, 2006.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. `http://www.netlib.org/lapack/lug/`.

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

[4] BLAS: Basic linear algebra subprograms. `http://www.netlib.org/blas/`.

[5] CUDA CUBLAS Library. http://developer.download.nvidia.com.

[6] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, 2008. IEEE Press.

[7] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] NVIDIA CUDA Compute Unified Device Architecture—Programming Guide. http://developer.download.nvidia.com, 2007.

[9] S. Tomov, R. Nath, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. UTK EECS Technical Report ut-cs-09-649, December 2009.

[10] S. Tomov and J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Technical Report 219, LAPACK Working Note 219, May 2009.

[11] James W. Demmel. Trading Off Parallelism and Numerical Stability, EECS Department, University of California, Berkeley, UCB/CSD-92-702, September 1992.

[12] Nicholas J. Higham. Stability of parallel triangular system solvers, *SIAM J. Sci. Comput.*, 16(2):400–413, 1995.

[13] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[14] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. Proceedings of the IEEE 93 (2005), no. 2, special issue on "Program Generation, Optimization, and Adaptation." Proceddings, vol. 93, 2, pp. 293–312.

[15] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. International Conference on Supercomputing, 1997, pp. 340–347.

[16] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, vol. 3, IEEE, 1998, pp. 1381–1384.

[17] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMMfor GPUs. In ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Michael Wolfe. Compilers and more: Optimizing GPU kernels. HPC Wire, http://www.hpcwire.com/features/33607434.html, October 2008.