

## Lecture #7: Memory Systems-II

Professor: S. Lennart Johnsson

TA: Wei Ding

## 1 Basic Linear Algebra operations

Most high performance floating-point processors, whether standard microprocessors or processors designed specifically for use in supercomputers, have a multiplier and an adder that can operate concurrently. The data for these operations is nearly always read from registers, and the results are written to registers. Some processors have a sufficient number of data paths to allow the multiplier and the adder to have completely separate operands, i.e., a total of six data paths between these two units and the registers. Other floating-point processors have a “multiply-accumulate” type architecture in which one of the inputs to the multiplier is the adder output. For either architecture, the peak performance can only be achieved when multiplication and addition is performed concurrently.

In our discussion of vector architectures and vectorization, we have already seen that the memory bandwidth required to match the computational bandwidth depends not only on the task to be performed, but also on how the elemental operations are scheduled. For instance, we saw that matrix-vector multiplication can be organized such that it suffices to retrieve one operand from memory for each multiply-add operation.

Historically, a set of primitive matrix operations known as the *Basic Linear Algebra Subroutines*, the BLAS, have been developed. These routines represent frequent operations in many scientific codes. Their efficient implementation has a significant impact on the performance of the overall code. The first BLAS [5], now known as the level-1 BLAS, were devised in the 1970s when vector architectures became common for supercomputers. `_AXPY` routines belong to the level-1 BLAS. But, these routines require large memory bandwidth for peak floating-point performance and a level-2 BLAS [3] for matrix operations represented by two nested loops in Fortran 77 or C, such as matrix-vector multiplication, evolved. The latest step in this evolution is the level-3 BLAS [1, 2] for matrix-matrix operations. This set of routines were mostly motivated by architectures having a memory hierarchy consisting of registers, one or more levels of intermediate memory (cache), and main memory. The ideas can be applied recursively and thus be extended to secondary and tertiary storage systems. The functions in the BLAS are summarized in Table 1.

### 1.1 Locality of reference

Scaling of a vector,  $y \leftarrow \alpha x$ , reads one scalar into a register, and then, for each component of the result, reads one element of the argument vector, multiplies it by the constant, and writes one result to memory. Thus, in 64-bit precision, each multiplication requires 8 bytes to be loaded

Level-1	Level-2	Level-3
$x \leftrightarrow y$	$y \leftarrow \alpha Ax + \beta y$	$C \leftarrow \alpha AB + \beta C$
$x \leftarrow \alpha x$	$y \leftarrow \alpha A^T x + \beta y$	$C \leftarrow \alpha A^T B + \beta C$
$y \leftarrow x$	$y \leftarrow \alpha A^H x + \beta y$	$C \leftarrow \alpha A^H B + \beta C$
$y \leftarrow \alpha x + y$	$y \leftarrow Ax$	$C \leftarrow \alpha AB^T + \beta C$
$dot \leftarrow x^T y$	$y \leftarrow A^T x$	$C \leftarrow \alpha AB^H + \beta C$
$dot \leftarrow x^H y$	$y \leftarrow A^H x$	$C \leftarrow \alpha A^T B^T + \beta C$
$dot \leftarrow \alpha + x^T y$	$y \leftarrow A^{-1} x$	$C \leftarrow \alpha A^H B^H + \beta C$
$nrm2 \leftarrow \ x\ _2$	$y \leftarrow A^{-T} x$	$C \leftarrow \alpha AA^T + \beta C$
$asum \leftarrow \ re(x)\ _1 + \ im(x)\ _1$	$y \leftarrow A^{-H} x$	$C \leftarrow \alpha AA^H + \beta C$
$amax \leftarrow 1^{st} k \ni  re(x_k)  +  im(x_k) $	$A \leftarrow \alpha xy^T + A$	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
	$A \leftarrow \alpha xy^H + A$	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$
	$A \leftarrow \alpha xx^H + A$	$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C$
	$A \leftarrow \alpha xy^T + y(\alpha x)^T + A$	$C \leftarrow \alpha A^H B + \alpha B^H A + \beta C$
	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	$B \leftarrow \alpha A^{-1} B$
		$B \leftarrow \alpha A^{-T} B$
		$B \leftarrow \alpha A^{-H} B$
		$B \leftarrow \alpha BA^{-1}$
		$B \leftarrow \alpha BA^{-T}$
		$B \leftarrow \alpha BA^{-H}$

Table 1: The Basic Linear Algebra Subroutines, the BLAS.

Operation	Mem. bandw. Bytes/flop
$y \leftarrow \alpha x$	16
$y \leftarrow \alpha x + y$	12
$dot \leftarrow x^T y$	8
$A \leftarrow \alpha x y^T + A$	8
$y \leftarrow \alpha A x + \beta y$	4
$C \leftarrow \alpha A B + \beta C$	8/b

Table 2: Memory bandwidth requirement in 64-bit-precision for full utilization of a floating-point unit with one adder and one multiplier.

into a register and 8 bytes to be stored from a register, or 16 bytes of memory bandwidth per floating-point operation. The inner-product routine,  $dot \leftarrow x^T y$ , reads two 8 byte quantities for each multiply-add operation, then stores one 8 byte result at the end. Thus, 16 bytes of memory bandwidth is required per 2 floating-point operations in 64-bit precision, or 8 bytes per floating-point operation. Both these routines belong to the level-1 BLAS.

We have already seen that matrix-vector multiplication can be organized such that it is sufficient to load one operand from memory for each multiply-add operation. Thus, in 64-bit precision, 8 bytes of memory bandwidth per floating-point operation suffice. Similarly, it is easy to show that the rank-1 update routine,  $A \leftarrow x y^T + A$ , requires a memory bandwidth of at least 8 bytes per floating-point operation. Though these bandwidth requirements are the same as for some of the level-1 BLAS, they are achieved after a careful scheduling of operations. The reduction in memory bandwidth demand is a factor of about two for most functions. Table 2 gives the memory bandwidth per floating-point operation demand for some of the BLAS. Most high performance computer systems are memory bandwidth limited, even with a careful choice of algorithm.

A level-3 BLAS routine performing the operation  $C \leftarrow A \times B + C$  allows for a further reduction in memory bandwidth requirement. It may be performed as a sequence of operations on  $b$  by  $b$  sub-blocks. If the blocks fit into the registers, then  $2b^3$  floating-point operations may be computed using  $3b^2$  input elements ( $b^2$  elements per operand), producing  $b^2$  results. If all contributions to a block of  $C$  are accumulated in registers, then it suffices to load  $2b^2$  inputs for each set of  $2b^3$  floating-point operations. All stores are delayed until all computations for a  $b \times b$  block of  $C$  are completed. Therefore,  $16b^2/2b^3$  bytes of memory bandwidth are required per floating-point operation in 64-bit precision, or  $8/b$  bytes/flop. Figure 1 illustrates the idea of block matrix multiplication. A rank- $b$  update of a matrix is equivalent to multiplication of a matrix of shape  $P \times b$  by a matrix of shape  $b \times R$ . The number of bytes per floating-point operation is  $12/b$ , a reduction in memory bandwidth requirement by a factor of  $3/(2b)$  compared to rank-1 updates.

The advantage of using a block matrix multiplication algorithm is clear. For  $b \times b$  blocks, at least  $3b^2$  registers are needed. Thus, for 64 registers, a block size  $b = 4$  is the best possible, whereas for 1024 registers a block size of  $b = 18$  may be possible. Compared to a DAXPY

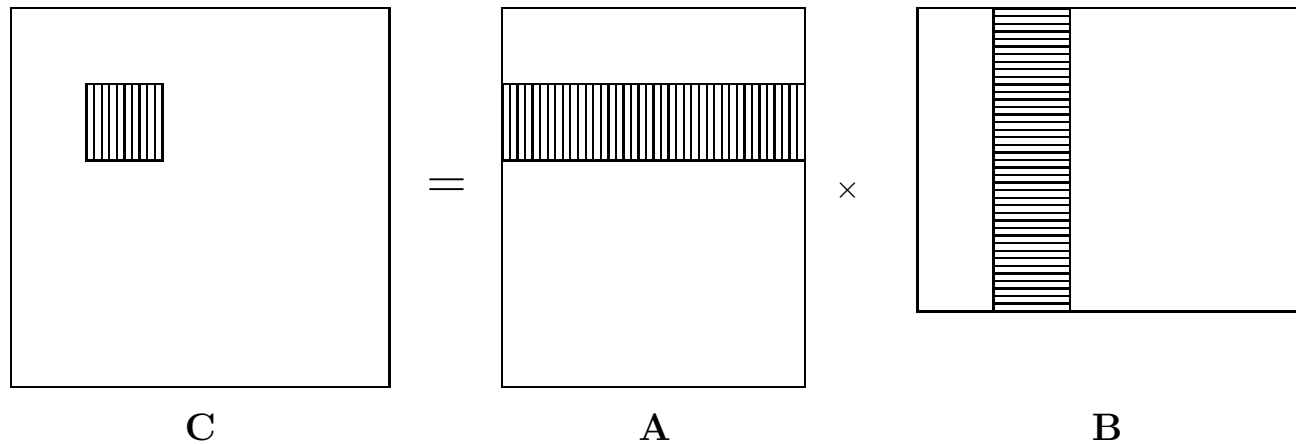


Figure 1: Block matrix multiplication.

based algorithm (D denotes 64-bit precision), the reduction in memory bandwidth requirement for  $b = 4$  is a factor of two, while for  $b = 18$  the reduction is a factor of 9. Thus, we conclude that for many matrix operations one memory reference per pair of floating-point operations is the best that can be expected. For more compute intensive operations, such as matrix–matrix multiplication requiring  $2Q$  operations per element for the multiplication of a  $P \times Q$  matrix by a  $Q \times R$  matrix, a factor of 10 improvement in demands for memory bandwidth is the best one can realistically expect from careful scheduling of operations with respect to register reuse. But, the idea of blocking can be used recursively for memory hierarchies with several levels.

Blocking at several levels is indeed used in some scientific subroutine libraries, such as the CMSSL [6]. The innermost blocking is based on the size of the register file, the second level is based on DRAM pages, and the third level on minimizing thrashing in Translation Lookahead Buffers, TLB. Moreover, if *multiple instances* of the same computation shall be performed, it is often advantages to include those in the blocking. By computing on multiple instances, it may be possible to further optimize the use of the memory hierarchy as well as increase vector lengths and minimize looping overhead.

Given the importance of conserving memory bandwidth it is of interest to determine whether the approaches discussed above indeed are the best possible, or if there may exist some other method that may offer even greater reductions. The block algorithms for matrix multiplication and the high radix FFT are indeed optimal for with respect to memory bandwidth use, as shown by Hong and Kung [4].

In a single processor system with a true random access memory, exploiting locality of reference beneficially in a memory hierarchy is a question of scheduling operations to allow for effective data reuse at various levels of the hierarchy. In distributed memory systems, where memory accesses require a different amount of time depending upon location, data allocation is important with respect to performance.

## References

- [1] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report Reprint No. 1, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.
- [2] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms: Model implementation and test programs. Technical Report Reprint No. 2, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.
- [3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. Technical Report Technical Memorandum 41, Argonne National Laboratories, Mathematics and Computer Science Division, November 1986.
- [4] J.W. Hong and H.T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the 13th ACM Symposium on the Theory of Computation*, pages 326–333. ACM, 1981.
- [5] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.
- [6] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.0*, 1992.