

Dense Linear Algebra Pattern

Name

Dense Linear Algebra

Problem

Some problems can be expressed in terms of linear algebraic operations over dense arrays of data (i.e. vectors and matrices containing mostly non-zero values). How does one balance data movement and computation in order to maximize performance?

Context

Many problems are expressed in terms of linear operations on vectors and matrices. Common examples include solving systems of linear equations, matrix factorization, eigenvalue problems, and matrix multiplication. The way in which a real-world problem is expressed in terms of these basic linear algebra tasks can be influenced by:

1. the structure of the problem
2. how a continuous problem is sampled
3. how a general solution is expanded in terms of simpler basis functions

The pivotal problem in computational linear algebra is how to balance data access costs with the cost of computation. Given the disparity between memory and CPU speeds, it is critical to maximize the amount of computation carried out for each item of data accessed from memory. For problems for which the systems of equations associated with a problem are dense (i.e. they contain mostly non-zero values), it is often possible to exploit a surface to volume effect providing plenty of computation to hide the costs of accessing data. How do we write software that can take advantage of this property in order to achieve high performance?

Forces

Universal forces

- A solution to this problem needs to trade off the cost of recomputing intermediate results with storing them and fetching them later.
- Overlapping computation with data movement requires software constructed around the details of a particular memory hierarchy. But this is in direct conflict with the need to write portable software.

Implementation dependent

- A linear algebra operation may dictate a distinct layout of data to achieve optimal performance. These data layouts, however, may conflict with those needed by other operations forcing the programmer to balance data copy overheads with potentially sub-optimal algorithms.

Solution

Linear Algebra Libraries

Dense linear algebra problems benefit from the fact that they map onto a relatively modest collection of standard mathematical operations such as the symmetric eigenvalue problem, multiplication, or LU factorization. Hence, for many operations in linear algebra, a linear algebra library such as LAPACK[1] or ScaLAPACK[2] for distributed memory systems may provide all that is needed. If this is the case, the task faced by the programmer is how to define his or her data so the related matrices are in the format required by the math library.

BLAS Routines and Templates

If a linear algebra library exists for your platform, it can save you a lot of effort; however, if such a library does not exist or you have specialized needs, using preexisting linear algebra subroutines to perform basic operations such as matrix-matrix multiplication can save time and yield good performance. Many of the more advanced dense linear algebra tasks are built from the simple routines contained in a standard BLAS (Basic Linear Algebra Subroutines) library. The Templates series of books [3][4], which are freely available electronically, contain extensive information which guides the reader through the construction and organization of various linear algebra solvers from the BLAS.

BLAS libraries are available for most architectures on the market and encapsulate hardware specific assembly code often required to achieve high performance. There are three levels of BLAS:

- BLAS Level 1: Vector/Vector operations, $O(N)$ for data movement and computation
- BLAS Level 2: Vector/Matrix operations, $O(N^2)$ for data movement and computation.
- BLAS Level 3: Matrix/Matrix operations, $O(N^2)$ for data movement, $O(N^3)$ for computation.

The BLAS level 3 routines stand out because of the smaller order of complexity for data movement ($O(N^2)$) than for computation ($O(N^3)$). Due to this fact, BLAS Level 3 routines can approach peak performance on most parallel systems. For this reason, a programmer would greatly benefit from structuring an algorithm to favor BLAS Level 3 functions. For example, replacing a large number of dot products (a BLAS level 1 function) with a single matrix-matrix multiplication (a BLAS level 3 function) can turn a problem bound by memory bandwidth into a compute-bound problem.

Auto-Tuned Libraries

If a BLAS library does not exist for the target architecture, there may still be hope. ATLAS (Automatically Tuned Linear Algebra Software) [5] and PHiPAC (Portable High Performance ANSI

C) [6] are two related projects which aim to automatically build and optimize BLAS routines for any architecture. Both rely on the concept of auto-tuning in order to achieve high performance without requiring any hand tuning or assembly coding. The auto-tuning process basically iterates through a large space of computational parameters and produces a library using the parameters which yielded the best performance on the target system.

Starting From Scratch

If none of the above suggestions are feasible, the task is too specialized, or for some reason you really want to start from scratch, it is important to understand how to structure computation in order to optimize utilization of the memory hierarchy. Because matrix multiplication is such an important building for many linear algebra routines, we will give an overview of its optimization here.

Naïve Approach

There are many ways to organize the computation of a basic matrix-matrix multiplication, written as:

$$\mathbf{C} = \mathbf{C} + \mathbf{A}\mathbf{B}$$

Naïve implementations carry out an inner product for each element of the product matrix. This is shown as 3 nested loops in Figure 1. The problem with this naïve approach is that it is highly suboptimal in terms of spatial locality. This is demonstrated in Figure 2 which shows how a matrix stored in column-major order will have adjacent row elements stored in separate cache lines. Accessing data in such a pattern can significantly increase data movement overhead.

```
% inner product approach
for i = 1:I
    for j = 1:J
        for k = 1:K
            C(i,j) = C(i,j) + A(i,k)*B(k,j);
```

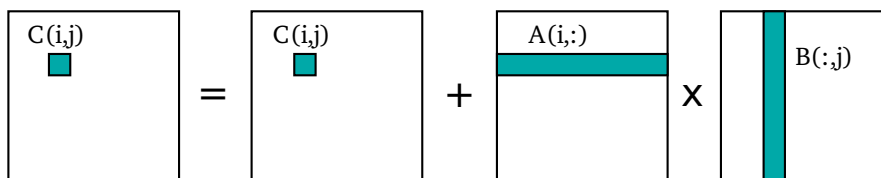


Figure 1: A naïve implementation of matrix multiply based on inner products. Each inner product computes a single element of the product matrix \mathbf{C} .

Outer Product Approach

A better approach is based on outer products (shown in Figure 3), in which the product of a column of \mathbf{A} and a row of \mathbf{B} acts as a rank one update to the entire product matrix \mathbf{C} . Notice that even

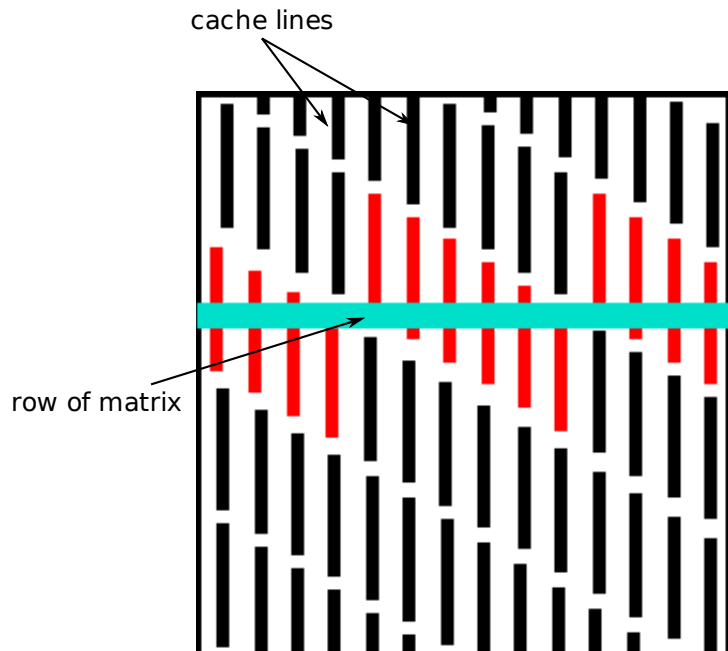


Figure 2: This diagram shows how a matrix stored in memory in column-major order is divided up into cache lines. As you can see, successive elements of a single row of the matrix (shown in blue) are stored in separate cache lines (each shown in red).

though the k loop has been moved to the outer most level, the three nested loops still produce the same result. While this approach doesn't solve the spatial locality problem, it can be utilized effectively in a blocked-based algorithm.

Blocked Approach

More sophisticated algorithms construct the product hierarchically from multiplications of smaller matrix blocks [7]. By computing outer products on small blocks of the input and output matrices, we can more effectively exploit spatial locality and data reuse. Figure 4 illustrates a blocking scheme with parameters I_0 , J_0 , and K_0 .

Auto-Tuning the Blocked Approach

When using a blocked approach, the optimal block sizes will change across architectures. Variables that can affect the best blocking parameters include the number of floating point registers, cache sizes, TLB size, time to access various levels of the memory hierarchy, number of floating point units, etc.. The interaction between all of these factors is so complicated that being able to decide on optimal blocking parameters while simply considering these variables is near impossible.

An effective way to arrive at optimal values for the blocking parameters, I_0 , J_0 , and K_0 , is through the use of auto-tuning. As mentioned above, ATLAS[5] and PHiPAC[6] are two examples

```

% outer product approach
for k = 1:K
  for i = 1:I
    for j = 1:J
      C(i,j) = C(i,j) + A(i,k)*B(k,j);
    
```

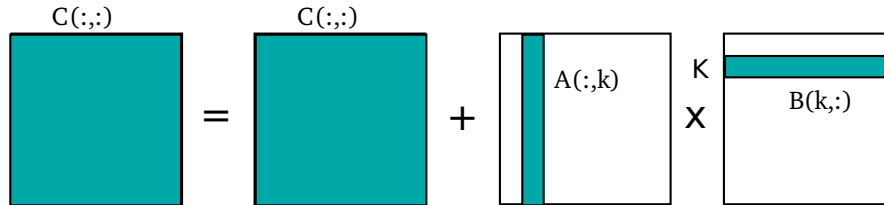


Figure 3: An implementation based on outer products. Each outer product represents a rank-one update to the entire product matrix.

of the successes of auto-tuning matrix multiply. In simple terms, to auto-tune a blocked matrix multiply, one would simply write a script to compile the blocked code with many different combinations of blocking parameters, measure the performance of each version of the code, and then choose the version that performed best.

One thing to be aware of when using a blocked approach is that many times, the optimal block size does not evenly divide the dimensions of the entire matrix. This can be handled in two ways. In the first method, one can simply zero pad the matrices up to a size that is divisible by the block size. Alternatively, functions that use smaller-sized blocks can be written to handle the “fringes”.

Multiple Level Blocking

A first level of blocking, as discussed above, is commonly referred to as “register blocking” because the data reuse properties of such an approach are thought to exploit the speed of the processor registers. In order to more fully exploit additional levels of the memory hierarchy, multiple levels of blocking can be used. This concept is illustrated at a high level in Figure 5.

Code for using an additional level of blocking would iterate through all of the larger, second-level blocks contained in each matrix. Within the code to handle each second-level block, the code to iterate through the first-level blocks contained within the current second-level block would be called.

Using a second level of blocking is thought to exploit the L1 cache which is slower than the register but is much larger in size. Utilizing a third level of blocking would target the L2 cache. Subsequent levels of blocking, while possibly effective, are not typically used.

Additional Optimizations

1. Explicit Loop Unrolling

While some loop unrolling can automatically be done by the compiler, explicitly coding loops as unrolled can allow the compiler to use additional optimizations, ignore false dependencies,

```

% blocked approach using outer products.

%iterate through blocks
for k = 1:K/K0
  for i = 1:I/I0
    Ablock = &A(i*I0,k*K0);
    for j = 1:J/J0
      Cblock = &C(i*I0,j*J0);
      Bblock = &B(k*K0,j*J0);
      do_block(Ablock,Bblock,Cblock);

void do_block(Ablock,Bblock,Cblock){
  %iterate through elements within a block
  for k0 = 1:K0
    for i0 = 1:I0
      for j0 = 1:J0
        Cblock(i0,j0) = Cblock(i0,j0) + Ablock(i0,k0)*Bblock(k0,j0);
}

```

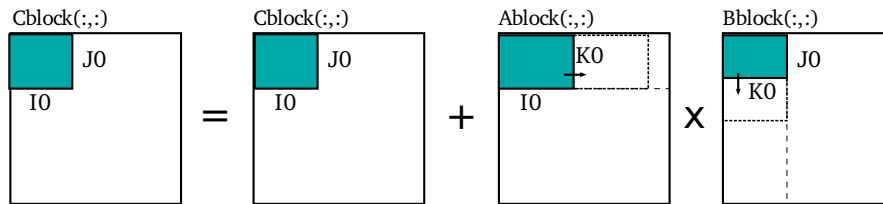


Figure 4: A blocked implementation, with blocking parameters I_0 , J_0 , and K_0 , based on outer products. Organizing computation in this way can greatly improve spatial locality.

and more effectively pipeline instructions. It is a good idea to fully unroll the loops in the within the function that handles the first-level blocking. Specific examples of explicit loop unrolling are shown in the PHiPAC paper[6]

2. Use local variables to explicitly remove false dependencies

If matrix values are read into locally declared variables and these local variables are used in the calculations, the compiler will allow the program to proceed without incorrectly assuming that it needs to wait for the result of a previous update to a matrix element. Example in [6]

3. Use SIMD instructions

Many architectures have vector instructions that can perform a single operation on multiple variable in a shorter amount of time than it would take to perform the operation on each variable individually. Matrix multiplication can greatly benefit from the use of such instructions.

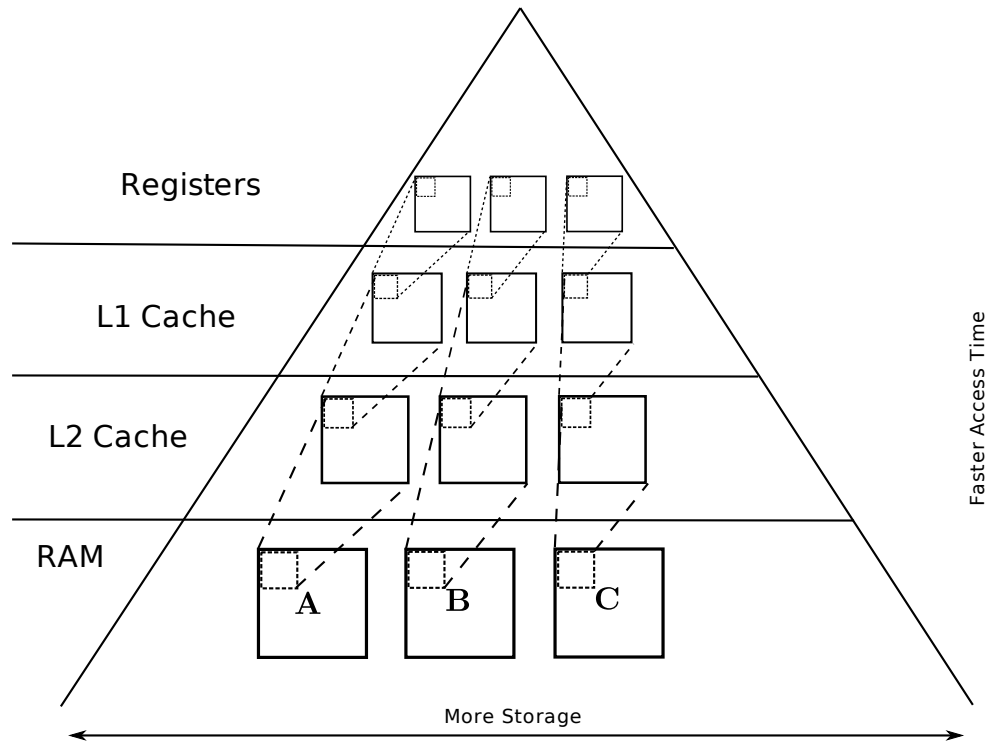


Figure 5: This diagram illustrated a multi-level blocking scheme formulated to exploit the multiple levels of memory hierarchy present in a typical modern microprocessor. Moving up the pyramid, memory latencies decrease, but storage size also decreases.

4. PHiPAC coding guidelines

Many more low level optimizations concerning pointer use, addressing, and reducing overall operations are covered in the PHiPAC paper.

Invariant

- The discrete representation of the problem (that leads to the matrices and vectors) accurately represents the original problem and produces well conditioned matrices.
- The final result approximates the mathematical definition to within tolerated round-off error.

Example

Computational Electromagnetics often leads to dense systems of linear equations. For example, in the late 1980s programs were developed to generate the radar cross section as a function of aircraft design parameters to reduce the signature of the aircraft (i.e. to support the design of stealth aircraft).

These simulations of electromagnetic fields use one of two numerical approaches:

1. Method of moments or boundary element method (frequency domain). This method produces large dense matrices.
2. Finite differences (time domain); which leads to large sparse matrices (which are sometimes solved by decomposing the sparse system into smaller dense matrices).

The numerical procedures are:

1. discretize the surface into triangular facets using standard modeling tools.
2. represent the amplitude of electromagnetic currents on the surface as unknown variables.
3. define integral equations over the triangular elements and discretize them as a set of linear equations.

At this point the integral equation is represented as the classic linear algebra problem:

$$\mathbf{Ax} = \mathbf{b}$$

where

\mathbf{A} is the (dense) impedance matrix,
 \mathbf{x} is the unknown vector of amplitudes, and
 \mathbf{b} is the excitation vector.

Now that the problem is defined in terms of a system of equations, it can be solved using one of the many preexisting solvers.

Known Uses

Dense linear algebra is heavily used throughout the computational sciences. In addition to the electro-magnetics problem we just described, problems from quantum mechanics (eigenvalue problems), statistics, computational finance and countless other problems are based on dense matrix computations. Sparse linear algebra is probably more common, but note that many sparse problems decompose into solutions over smaller dense matrices.

Related Patterns

This pattern is closely related to a number of lower level patterns depending on the direction taken in constructing the detailed solution. Often, the geometric decomposition pattern is used in constructing parallel dense linear algebra problems. For matrices represented as hierarchical data structures, the divide and conquer pattern is indicated. The data structures used with dense linear algebra problems especially on NUMA or distributed memory systems are addressed in the Distributed arrays pattern. And finally, depending on the details of how a problem is represented, a problem can result in sparse instead of dense matrices (and hence the Sparse Linear Algebra pattern).

References

References

- [1] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen, “LAPACK: A portable linear algebra library for high-performance computers,” in *Supercomputing’90. Proceedings of*, 1990, pp. 2–11.
- [2] J. Choi, J. Dongarra, R. Pozo, and D. Walker, “ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers,” in *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, 1992, pp. 120–127.
- [3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.
- [4] Z. Bai, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial Mathematics, 2000.
- [5] R. Whaley and J. Dongarra, “Automatically tuned linear algebra software,” in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society Washington, DC, USA, 1998, pp. 1–27.
- [6] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel, “Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology,” in *Proceedings of the 11th international conference on Supercomputing*. ACM Press New York, NY, USA, 1997, pp. 340–347.
- [7] R. Van De Geijn and J. Watts, “SUMMA: scalable universal matrix multiplication algorithm,” *Concurrency Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

Authors

Eric Battenberg, Tim Mattson, Dai Bui

Sherherd

Mark Murphy