DATA ACCESS OPTIMIZATION

Dr Noor Mahammad Sk

Center for High Performance Reconfigurable Computing

Hprcse

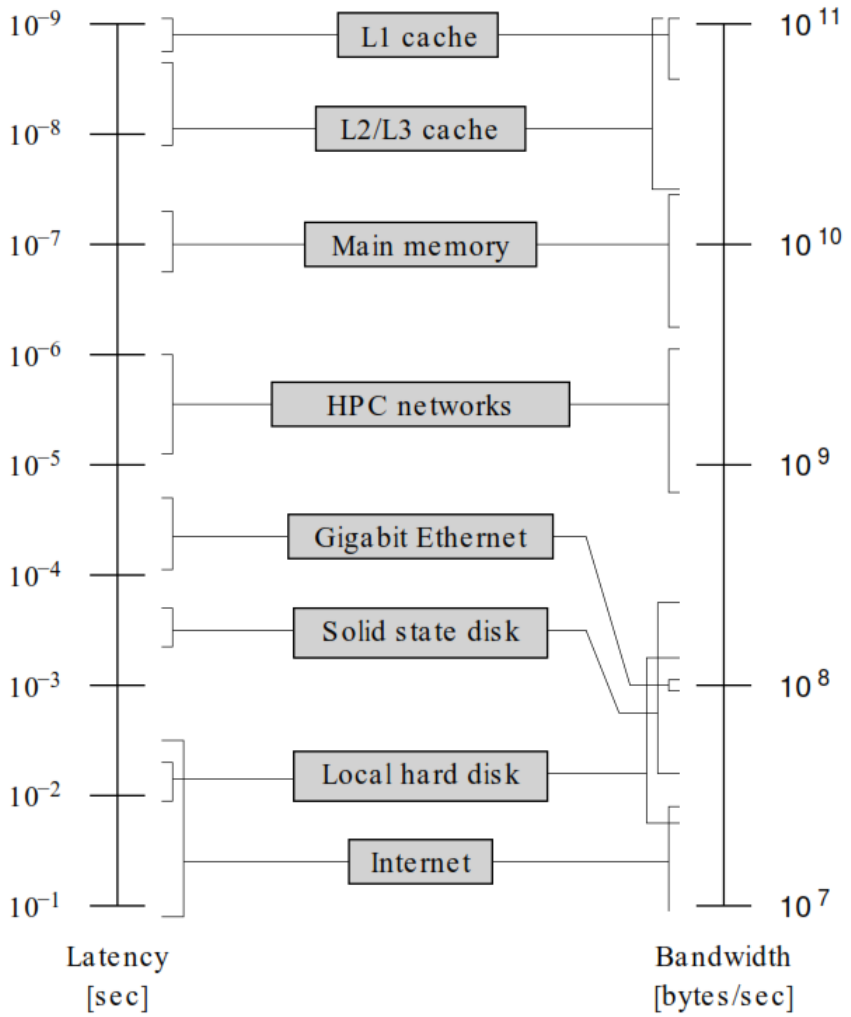
Introduction

- Data access is one of the performance-limiting factors in HPC.
- Microprocessors tend to be inherently “unbalanced” with respect to the relation of theoretical peak performance versus memory bandwidth.
- Many applications in science and engineering consist of loop-based code that moves large amounts of data in and out of the CPU.
- On-chip resources tend to be underutilized and performance is limited only by the relatively slow data paths to memory or even disks.

Bandwidth vs Latency

- Bandwidth is the amount of data that can be transferred from one point to another normally measured in seconds.
- Latency is the time that a data packet takes to travel from one point to another.

Overview of several data paths present in modern parallel computer systems



- The functional units that perform the computational work, sit at the top of this hierarchy.
- In terms of bandwidth, the slowest data paths are three to four orders of magnitude away
- Eight in terms of latency.
- The deeper a data transfer must reach down through the different levels in order to obtain required operands for some calculation, the harder the impact on performance.
- Any optimization attempt should aim at reducing traffic over slow data paths, or,
- Should at least make data transfer as efficient as possible..

Bandwidth-based Performance Modeling

- If the program at hand is already using the resources in the best possible way
- One can often estimate the theoretical performance of loop-based code that is bound by bandwidth limitations by simple rules of thumb.
- The central concept to introduce here is *balance*.
- *The machine balance B_m* of a processor chip is the ratio of possible memory bandwidth in GWords/sec to peak performance in GFlops/sec:

$$B_{\mathrm{m}} = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak performance [GFlops/sec]}} = \frac{b_{\max}}{P_{\max}}$$

Example

- Consider a dual-core chip with a clock frequency of 3.0 GHz that can perform at most four flops per cycle (per core)
- Has a memory bandwidth of 10.6 GBytes/sec. (64 bit word)
- This processor would have a machine balance of 0.055 W/F.
- At the time of writing, typical values of B_m lie in the range between 0.03 W/F for standard cache-based microprocessors and 0.5 W/F for top of the line vector processors.

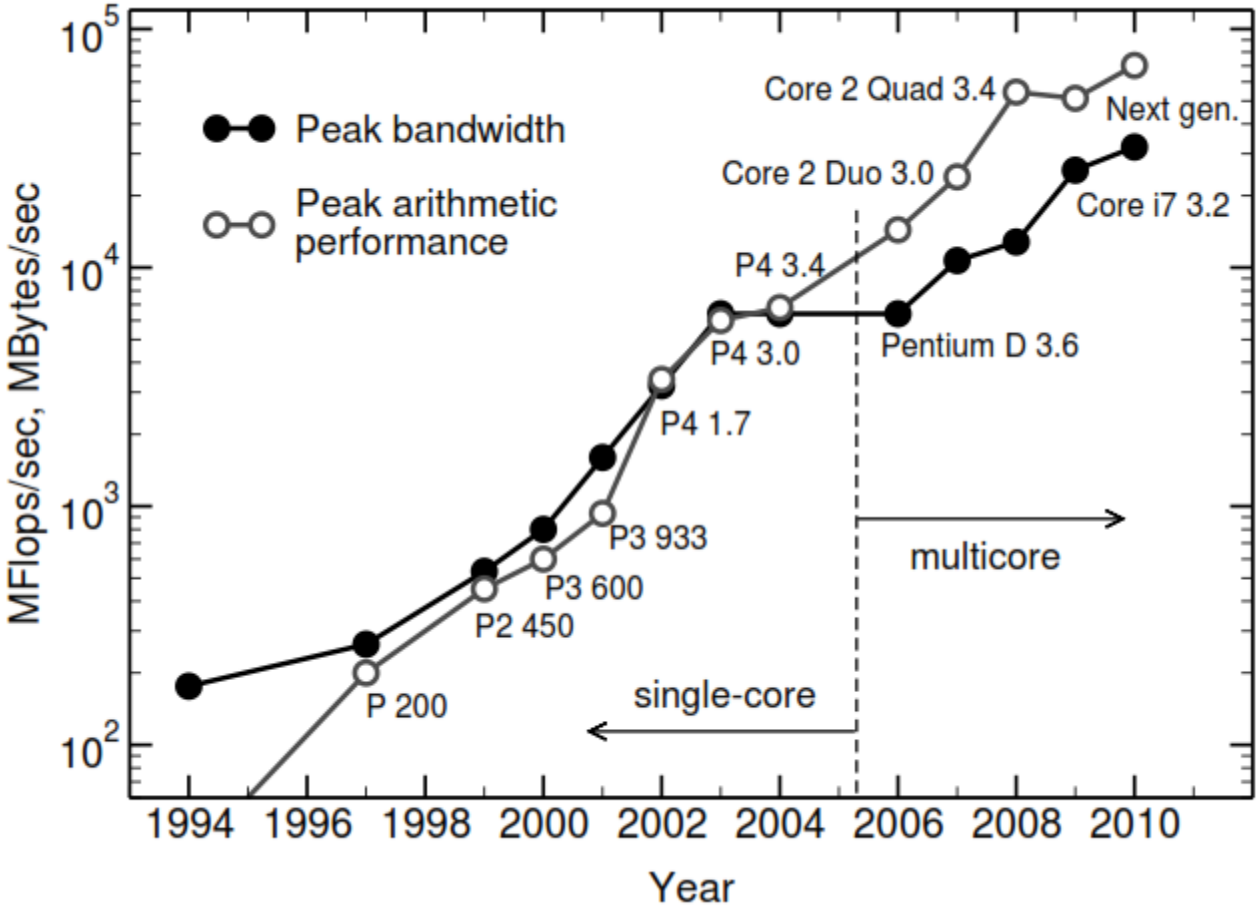
Balance Values

- Typical balance values for operations limited by different transfer paths.
- In case of network and disk connections, the peak performance of typical dual-socket compute nodes was taken as a basis.

data path	balance [W/F]
cache	0.5–1.0
machine (memory)	0.03–0.5
interconnect (high speed)	0.001–0.02
interconnect (GBit ethernet)	0.0001–0.0007
disk (or disk subsystem)	0.0001–0.01

Peak Performance and Memory Bandwidth

- Peak performance did grow faster than memory bandwidth before 2005
- The introduction of the first dual-core chip (Pentium D) really widened the DRAM gap considerably.



Light speed of a Loop

- "Data traffic" refers to all words transferred over the performance-limiting data path
- This metric dependent on the hardware.
- The reciprocal of code balance is often called *computational intensity.*
- The expected maximum fraction of peak performance of a code with balance *B*c, on a machine with balance *B*m:

$$l = \min\left(1, \frac{B_\mathrm{m}}{B_\mathrm{c}}\right)$$

- We call this fraction the *lightspeed of a loop*
- Performance in GFlops/sec is

$$P = lP_\mathrm{max} = \min\left(P_\mathrm{max}, \frac{b_\mathrm{max}}{B_\mathrm{c}}\right)$$

- If $l \simeq 1$, *performance* is not limited by bandwidth but other factors, either inside the CPU or elsewhere.

Simple performance model is based on some Crucial assumptions:

- The loop code makes use of all arithmetic units (MULT and ADD) in an optimal way.
- If this is not the case one must introduce a correction factor that reflects the ratio of “effective” to absolute peak performance
- E.g., if only ADDs are used in the code, effective peak performance would be half of the absolute maximum.
- Similar considerations apply if less than the maximum number of cores per chip are used.
- The loop code is based on double precision floating-point arithmetic.
- Data transfer and arithmetic overlap perfectly.
- The slowest data path determines the loop code’s performance.
- All faster data paths are assumed to be infinitely fast.
- The system is in “throughput mode,” i.e., latency effects are negligible.

Simple performance model is based on some Crucial assumptions:

- It is possible to saturate the memory bandwidth that goes into the calculation of machine balance to its full extent.
- Recent multicore designs tend to underutilize the memory interface if only a fraction of the cores use it.
- This makes performance prediction more complex, since there is a separate “effective” machine balance that is not just proportional to N^{-1} for each core count N.

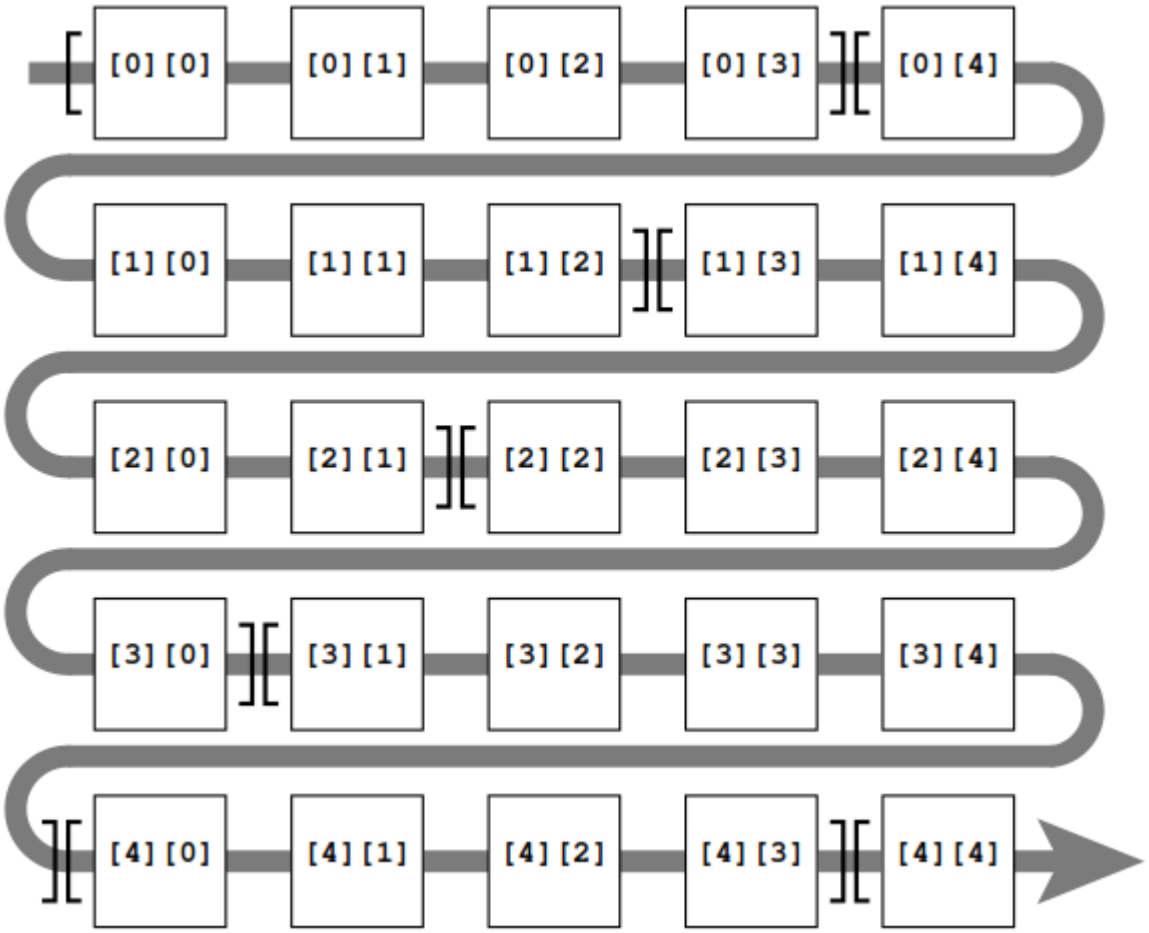
Read and Write Balance

- Maximum bandwidth is often not available in both directions (read and write) concurrently.
- It may be the case, e.g., that the relation from maximum read to maximum write bandwidth is 2:1.
- A write stream cannot utilize the full bandwidth in that case.
- Protocol overhead, deficiencies in chipsets, error correcting memory chips, and large latencies all cut on available bandwidth.
- Data paths inside the processor chip, e.g., connections between L1 cache and registers, can be unidirectional.
- If the code is not balanced between read and write operations, some of the bandwidth in one direction is unused.
- This should be taken into account when applying balance analysis for in-cache situations.

Storage order

- Multidimensional arrays, first and foremost matrices or matrix-like structures, are omnipresent in scientific computing.
- Data access is a crucial topic here as the mapping between the inherently one-dimensional, cache line based memory layout of standard computers.
- Any multidimensional data structure must be matched to the order in which code loads and stores data so that spatial and temporal locality can be employed.
- *Strided access to a one-dimensional array reduces spatial locality,* leading to low utilization of the available bandwidth.

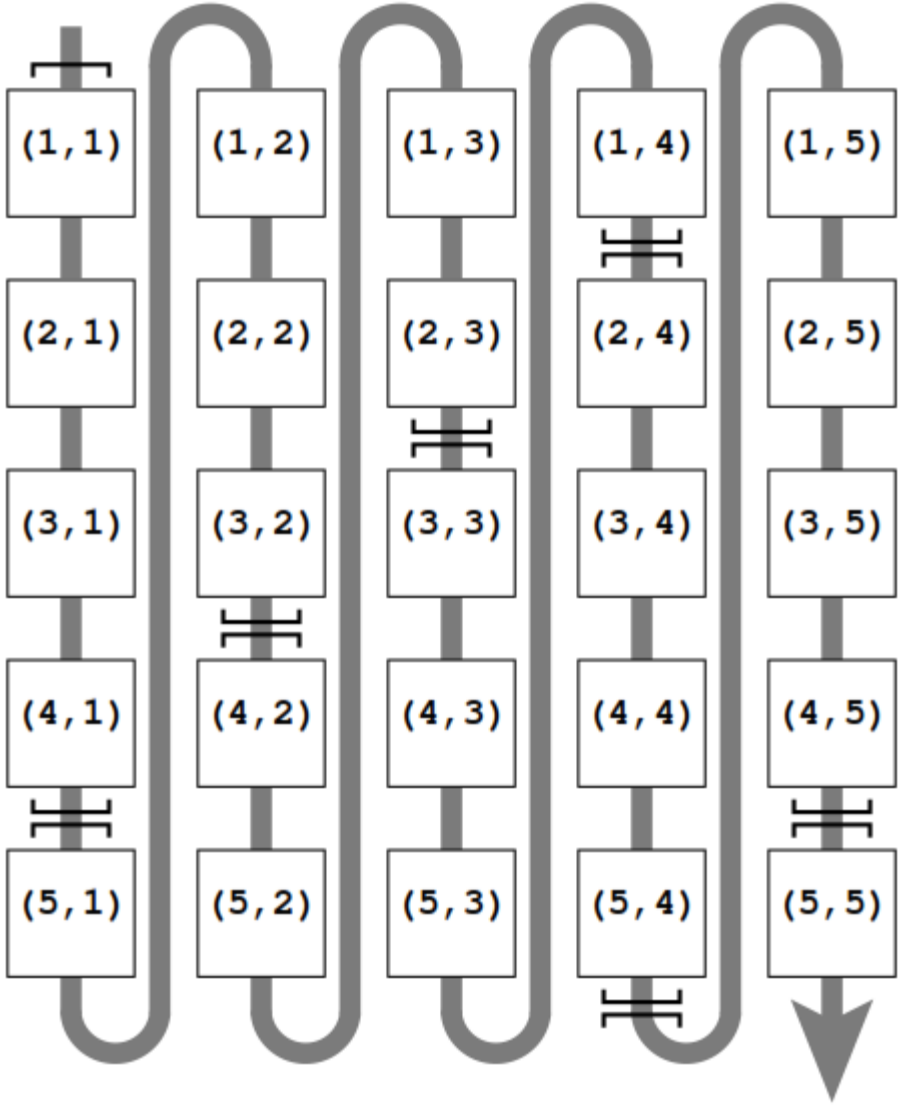
Multidimensional Arrays Access Patterns




```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = i*j;
  }
}
```

- Row major order matrix storage scheme, as used by the C programming language.
- Matrix rows are stored consecutively in memory.
- Cache lines are assumed to hold four matrix elements and are indicated by brackets.

Multidimensional Arrays Access Patterns




```
1 do i=1,N
2    do j=1,N
3       A(i,j) = i*j
4    enddo
5 enddo
```

- Column major order matrix storage scheme, as used by the Fortran programming language.
- Matrix columns are stored consecutively in memory.
- Cache lines are assumed to hold four matrix elements and are indicated by brackets.

Algorithm Classification and Access Optimizations

- The optimization potential of many loops on cache-based processors can easily be estimated just by looking at basic parameters like
  - The scaling behavior of data transfers
  - Arithmetic operations versus problem size.
- It can then be decided whether investing optimization effort would make sense.

O(N)/O(N)

- ***Optimization* very limited:**
- If both the number of arithmetic operations and the number of data transfers (loads/stores) are proportional to the problem size (or "loop length") *N*.
- Examples:
- Scalar products
- Vector additions, and
- Sparse matrix vector multiplication

- They are inevitably memory-bound for large *N*
- *Compiler-generated code achieves good performance* because *O(N)/O(N) loops tend to be quite simple and the correct software pipelining* strategy is obvious.

Loop Fusion

- Even if loops are not nested there is sometimes room for improvement

```
1  do i=1,N
2    A(i) = B(i) + C(i)
3  enddo
4  do i=1,N
5    Z(i) = B(i) + E(i)
6  enddo
```

loop fusion →

```
! optimized
do i=1,N
  A(i) = B(i) + C(i)
! save a load for B(i)
  Z(i) = B(i) + E(i)
enddo
```

- *Loop fusion has achieved an O(N) data reuse for the two-loop constellation so* that a complete load stream could be eliminated.
- This kind optimization is applied by compilers by themselves

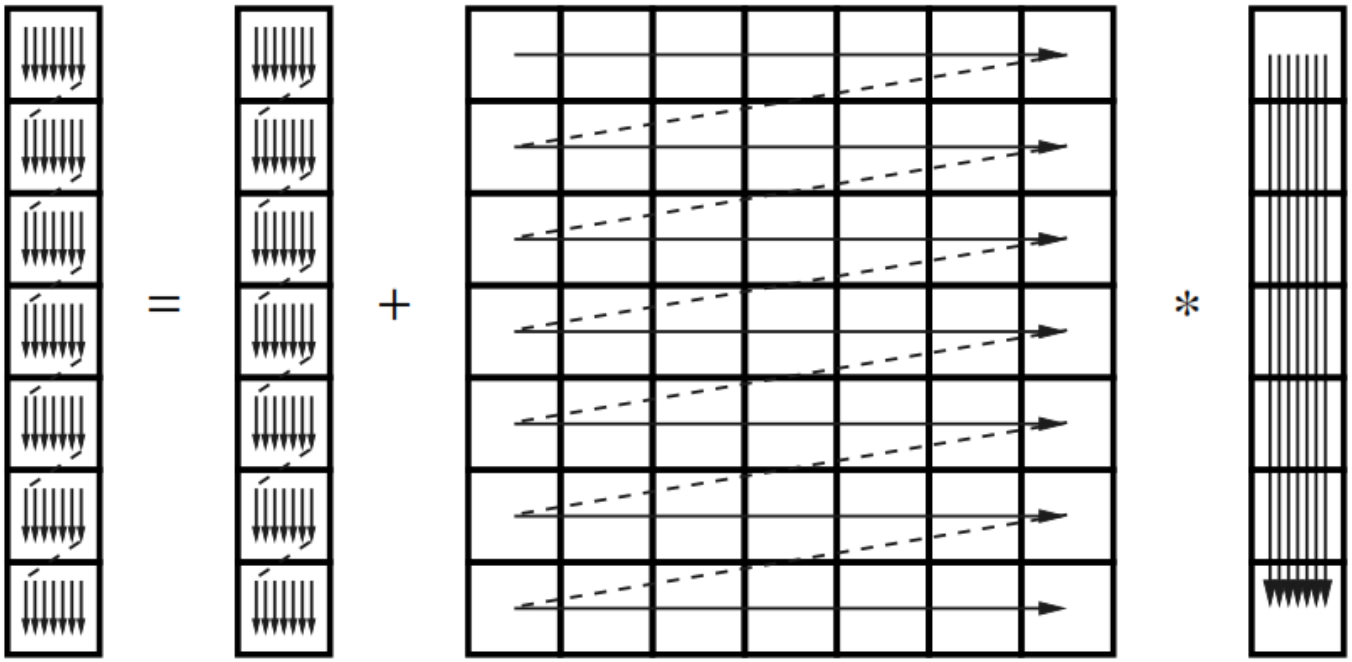
$O(N^2)/O(N^2)$

- In typical two-level loop nests where each loop has a trip count of *N, there are* $O(N^2)$ operations for $O(N^2)$ loads and stores.
- Examples are dense matrix-vector multiply, matrix transpose, matrix addition, etc.
- Although the situation on the inner level is similar to the *O(N)/O(N) case and the problems are generally memory-bound, the* nesting opens new opportunities.
- Optimization, however, is again usually limited to a constant factor of improvement

Unoptimized *N* ×*N dense matrix vector multiply*

- The RHS vector is loaded *N* times.
- As an example we consider dense matrix-vector multiply (MVM)

```
1  do i=1,N
2    tmp = C(i)
3    do j=1,N
4      tmp = tmp + A(j,i) * B(j)
5    enddo
6    C(i) = tmp
7  enddo
```



- This code has a balance of 1 W/F (two loads for A and B and two flops).
- Array C is indexed by the outer loop variable, so updates can go to a register and do not count as load or store streams.
- Matrix A is only loaded once, but B is loaded *N times, once for each outer loop iteration.*
- One would like to apply the same fusion trick as above, but there are not just two but *N inner loops to fuse.*

Loop Unrolling

```
1  ! remainder loop
2  do r=1,mod(N,m)
3    do j=1,N
4      C(r) = C(r) + A(j,r) * B(j)
5    enddo
6  enddo
7  ! main loop
8  do i=r,N,m
9    do j=1,N
10     C(i) = C(i) + A(j,i) * B(j)
11   enddo
12   do j=1,N
13     C(i+1) = C(i+1) + A(j,i+1) * B(j)
14   enddo
15   ! m times
16   ...
17   do j=1,N
18     C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
19   enddo
20 enddo
```

- *The outer loop is traversed* with a stride *m and the inner loop is replicated m times.*
- *We thus have to deal with* the situation that the outer loop count might not be a multiple of *m.*
- *This case has to* be handled by a remainder loop:

Unroll and jam

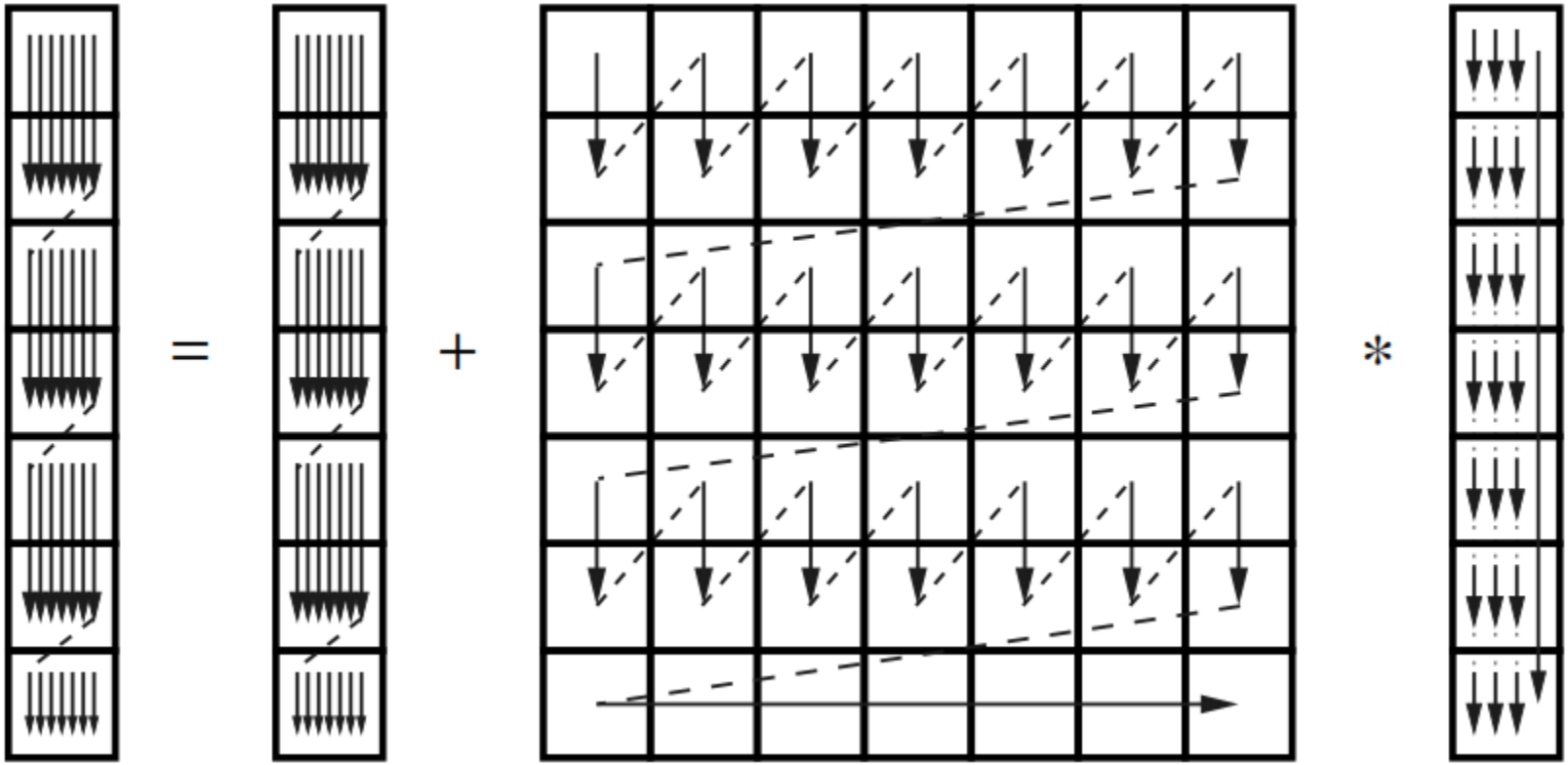
```
1  ! remainder loop ignored
2  do i=1,N,m
3    do j=1,N
4      C(i) = C(i) + A(j,i) * B(j)
5      C(i+1) = C(i+1) + A(j,i+1) * B(j)
6      !  m times
7      ...
8      C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
9    enddo
10 enddo
```

unroll and jam

- The combination of outer loop unrolling and fusion is often called *unroll and jam.*
- *By m-way unroll and jam we have achieved an m-fold reuse of each element of B from* register so that code balance reduces to (*m+1)/2m which is clearly smaller than one* for *m > 1.*
- *If m is very large, the performance gain can get close to a factor of two.*
- *In* this case array B is only loaded a few times or, ideally, just once from memory.
- As A is always loaded exactly once and has size N^2, the total memory traffic with *m-way* unroll and jam amounts to $N^2(1 + 1/m) + N$.

The memory access pattern for two-way unrolled dense matrix-vector multiply

- Two-way unrolled dense matrix vector multiply.
- The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two.
- The remainder loop is only a single (outer) iteration in this example.



Loop Unroll

- Assumes, that register pressure is not too large,
- i.e., the CPU has enough registers to hold all the required operands used inside the now quite sizeable loop body.
- If this is not the case, the compiler must spill register data to cache, slowing down the computation.
- Unroll and jam can be carried out automatically by some compilers at high optimization levels.

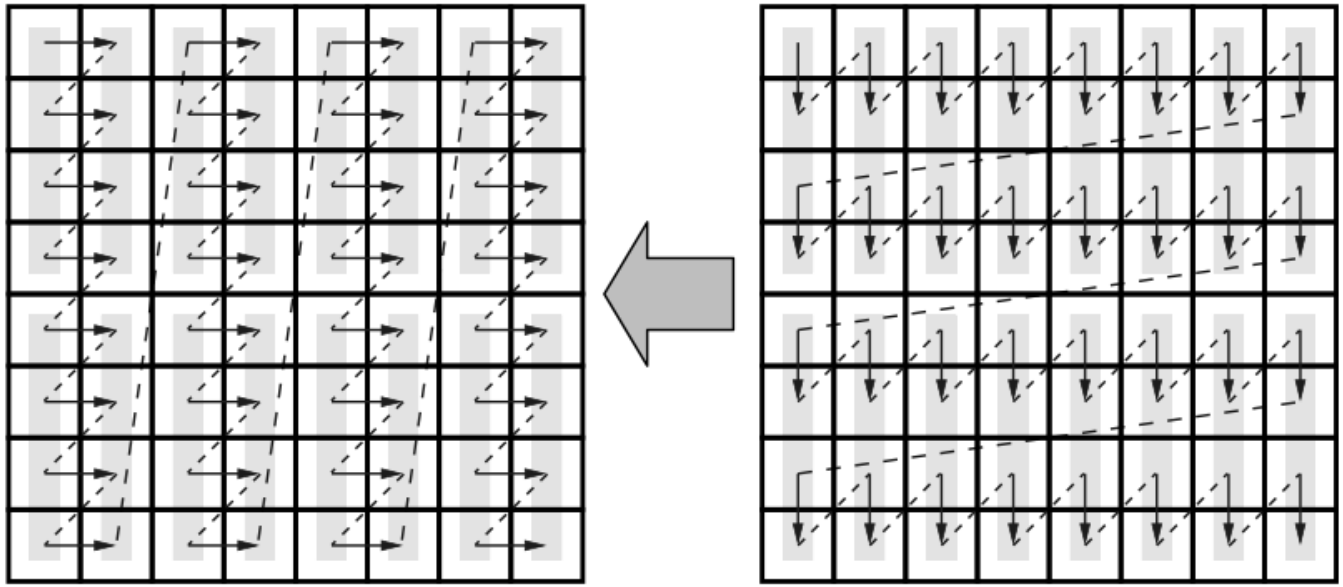
Matrix Transpose Code

- The matrix transpose code is another typical example for an $O(N^2)/O(N^2)$ problem
- There is no direct opportunity for saving on memory traffic;
- Both matrices have to be read or written exactly once.
- Nevertheless, by using unroll and jam on the "flipped" version a significant performance boost of nearly 50% is observed

```
1  do j=1,N,m
2    do i=1,N
3      A(i,j)      = B(j,i)
4      A(i,j+1)    = B(j+1,i)
5      ...
6      A(i,j+m-1)  = B(j+m-1,i)
7    enddo
8  enddo
```

Two-way unrolled "flipped" matrix transpose

- Naively one would not expect any effect at *m = 4 because the basic analysis stays* the same:
- In the mid-*N region the number of available cache lines is large enough* to hold up to *L* columns of the store stream.
- Figure shows the situation for *m = 2.*
- *However, the fact that m words in each of the load stream's cache lines are* now accessed *in direct succession reduces the TLB misses by a factor of m, although* the TLB is still way too small to map the whole working set.



- Cutting down on TLB misses does not remedy the performance breakdown for large *N when the cache gets too small to hold N cache lines.*
- *It would* be nice to have a strategy which reuses the remaining *L-m words of the strided* stream's cache lines right away so that each line may be evicted soon and would not have to be reclaimed later.
- A "brute force" method is L_c-way unrolling, but this approach leads to large-stride accesses in the store stream and is not a general solution as large unrolling factors raise register pressure in loops with arithmetic operations.

Loop Blocking

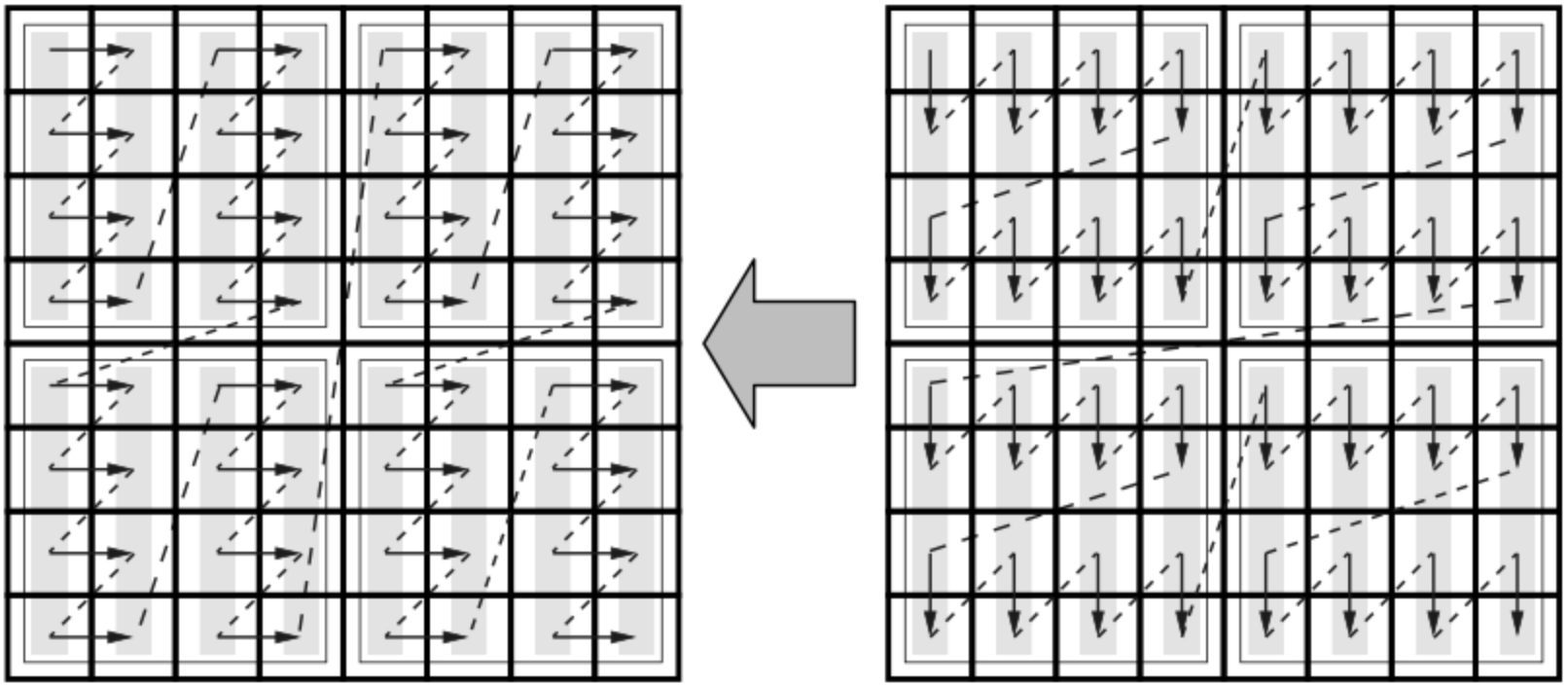
- *Loop blocking can achieve optimal cache line use without additional register pressure.*
- It does not save load or store operations but increases the cache hit ratio.
- For a loop nest of depth *d, blocking introduces up to d additional outer loop levels that cut* the original inner loops into chunks:

```
1  do jj=1,N,b
2    jstart=jj; jend=jj+b-1
3    do ii=1,N,b
4      istart=ii; iend=ii+b-1
5      do j=jstart,jend,m
6        do i=istart,iend
7          a(i,j) = b(j,i)
8          a(i,j+1) = b(j+1,i)
9          ...
10         a(i,j+m-1) = b(j+m-1,i)
11       enddo
12     enddo
13   enddo
14 enddo
```

Blocking

- In this example we have used *two-dimensional blocking with identical blocking factors b for both loops in addition to m-way unroll and jam.*
- *This change does not alter* the loop body so the number of registers needed to hold operands stays the same.
- However, the cache line access characteristics are much improved
- A combination of two-way unrolling and 4 ×4 blocking.
- If the blocking factors are chosen appropriately, the cache lines of the strided stream will have been used completely at the end of a block and can be evicted “soon.”
- Hence, we expect the large-*N performance breakdown to disappear.*

4 ×4 blocked and two-way unrolled “flipped” matrix transpose



Loop Blocking

- Loop blocking is a very general and powerful optimization that can often not be performed by compilers.
- The correct blocking factor to use should be determined experimentally through careful benchmarking, but one may be guided by typical cache sizes,
- i.e., when blocking for L1 cache the aggregated working set size of all blocked inner loop nests should not be much larger than half the cache.
- Which cache level to block for depends on the operations performed and there is no general recommendation.

$O(N^3)/O(N^2)$

- If the number of operations is larger than the number of data items by a factor that grows with problem size
- By the techniques described above (unroll and jam, loop blocking) it is sometimes possible for these kinds of problems to render the implementation cache-bound.
- Examples for algorithms that show $O(N^3)/O(N^2)$ characteristics are dense matrix-matrix multiplication (MMM) and dense matrix diagonalization.
- It is beyond the scope of this book to develop a well-optimized MMM,
- Let alone eigenvalue calculation, but we can demonstrate the basic principle by means of a simpler example which is actually of the *$O(N^2)/O(N^2)$ type:*

```
1 do i=1,N
2   do j=1,N
3     sum = sum + foo(A(i),B(j))
4   enddo
5 enddo
```

$O(N^3)/O(N^2)$

- The complete data set is *O(N) here but* $O(N^2)$ operations (calls to foo(), additions) are performed on it.
- In the form shown above, array B is loaded from memory *N* times, so the total memory traffic amounts to *N(N +1) words.*
- *m-way unroll and jam* is possible and will immediately reduce this to *N(N/m+1), but the disadvantages* of large unroll factors have been pointed out already.
- Blocking the inner loop with a blocksize of *b, however,* has two effects:
- Array B is now loaded only once from memory, provided that *b is small enough* so that *b elements fit into cache and stay there as long as they are needed.*
- Array A is loaded from memory *N/b times instead of once.*

```
1 do jj=1,N,b
2   jstart=jj; jend=jj+b-1
3   do i=1,N
4     do j=jstart,jend
5       sum = sum + foo(A(i),B(j))
6     enddo
7   enddo
8 enddo
```

- Although A is streamed through cache *N/b times,*
- *The probability that the current* block of B will be evicted is quite low
- The reason being that those cache lines are used very frequently and thus kept by the LRU replacement algorithm.
- This leads to an effective memory traffic of *N(N/b+1) words.*
- *As b can be made much larger than* typical unrolling factors, blocking is the best optimization strategy here.
- Unroll and jam can still be applied to enhance in-cache code balance.
- The basic N^2 dependence is still there, but with a prefactor that can make the difference between memory bound and cache-bound behavior.
- A code is cache-bound if main memory bandwidth and latency are not the limiting factors for performance any more.
- Whether this goal is achievable on a certain architecture depends on the cache size, cache and memory speeds, and the algorithm, of course.

- Algorithms of the $O(N^3)/O(N^2)$ type are typical candidates for optimizations that can potentially lead to performance numbers close to the theoretical maximum.
- If blocking and unrolling factors are chosen appropriately, dense matrix-matrix multiply, e.g., is an operation that usually achieves over 90% of peak for *N ×N matrices* if *N is not too small.*
- *It is provided in highly optimized versions by system vendors* as, e.g., contained in the BLAS (Basic Linear Algebra Subsystem) library.
- One might ask why unrolling should be applied at all when blocking already achieves the most important task of making the code cache-bound.
- The reason is that even if all the data resides in a cache, many processor architectures do not have the capability for sustaining enough loads and stores per cycle to feed the arithmetic units continuously.
- For instance, the current x86 processors from Intel can sustain one load and one store operation per cycle, which makes unroll and jam mandatory if the kernel of a loop nest uses more than one load stream, especially in cache-bound situations like the blocked $O(N^2)/O(N)$ *example above.*

Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.