Module 1
Effective use of a single core

Slides are based on Chapters 1,2,3 of the following book:

Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Overview of Module 1

- A *quick overview* of the architecture of modern cache-based microprocessors
- "Common sense" optimization techniques
- Data access optimization

Cache-based microprocessors

"Stored-program computer"

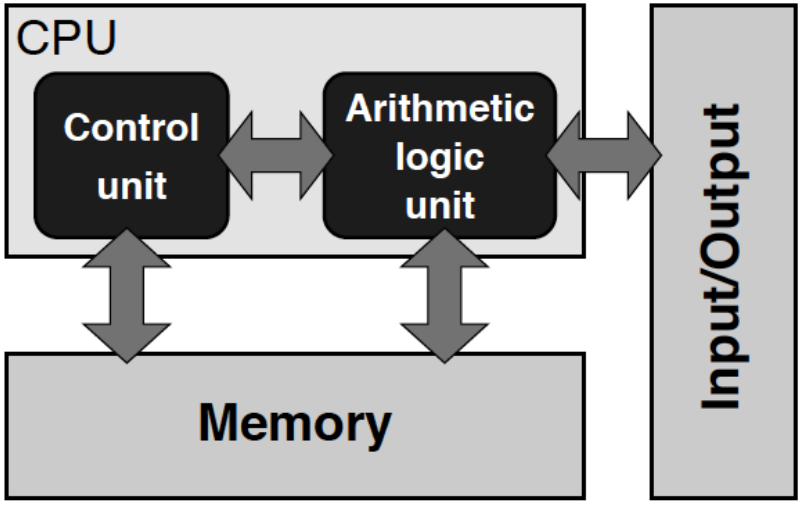


Figure 1.1: Stored-program computer architectural concept. The "program," which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.

Cache-based microprocessor

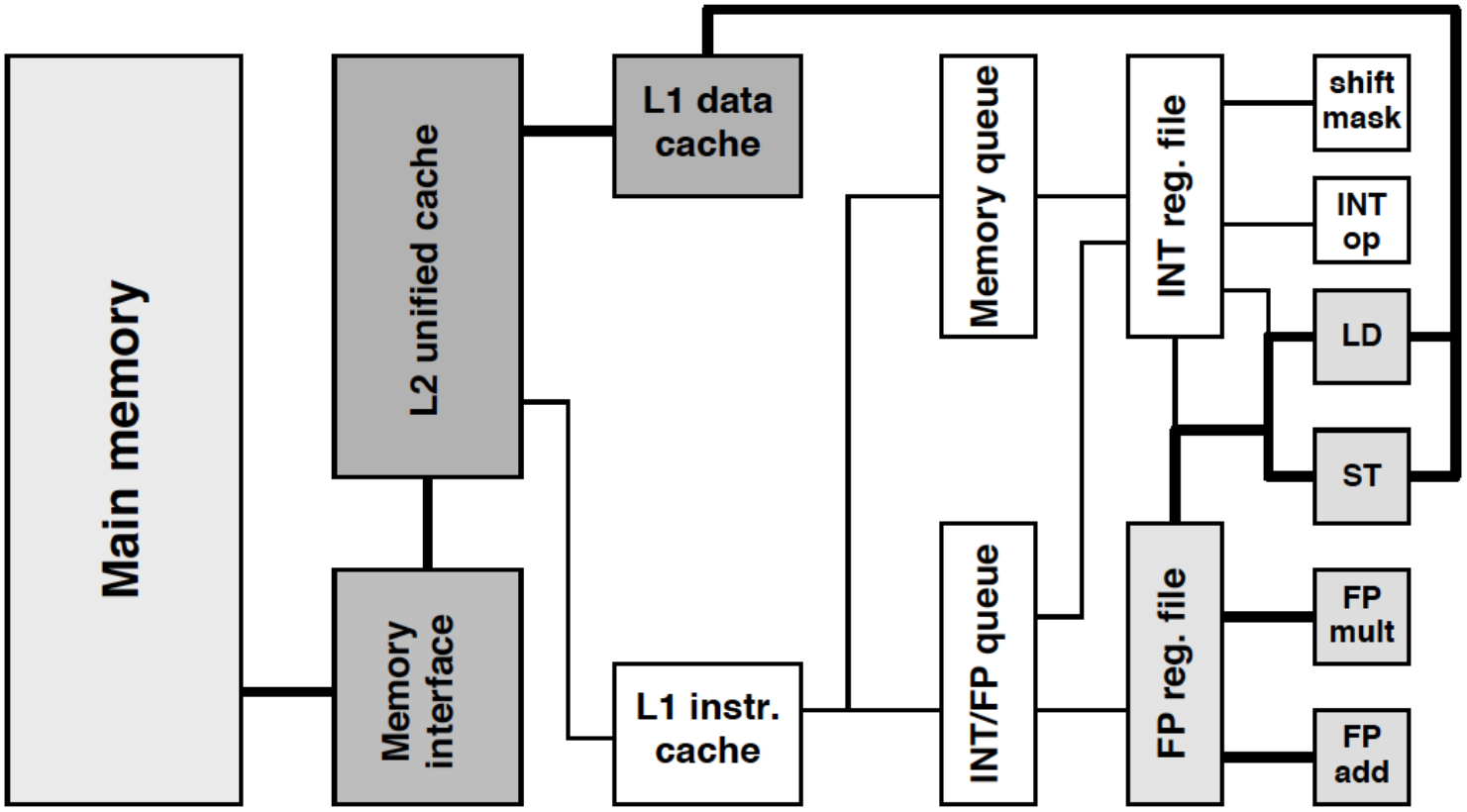


Figure 1.2: Simplified block diagram of a typical cache-based microprocessor (one core). Other cores on the same chip or package (socket) can share resources like caches or the memory interface. The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.

Pipelined functional units

Subdividing complex operations into simple components that can be executed using different functional units, it is possible to increase *instruction throughput*—the number of instructions executed per clock cycle.

This is the most elementary example of *instruction-level parallelism* (ILP).

Optimally pipelined execution leads to a throughput of one instruction per cycle per pipeline.

SIMD

The SIMD (single-instruction-multiple-data) concept became widely known with the first vector supercomputers in 1970s.

Modern cache-based processors have instruction set extensions for both integer and floating-point operations. They allow the concurrent execution of arithmetic operations on “wide” registers, each holding multiple numerical values.

Example of SIMD

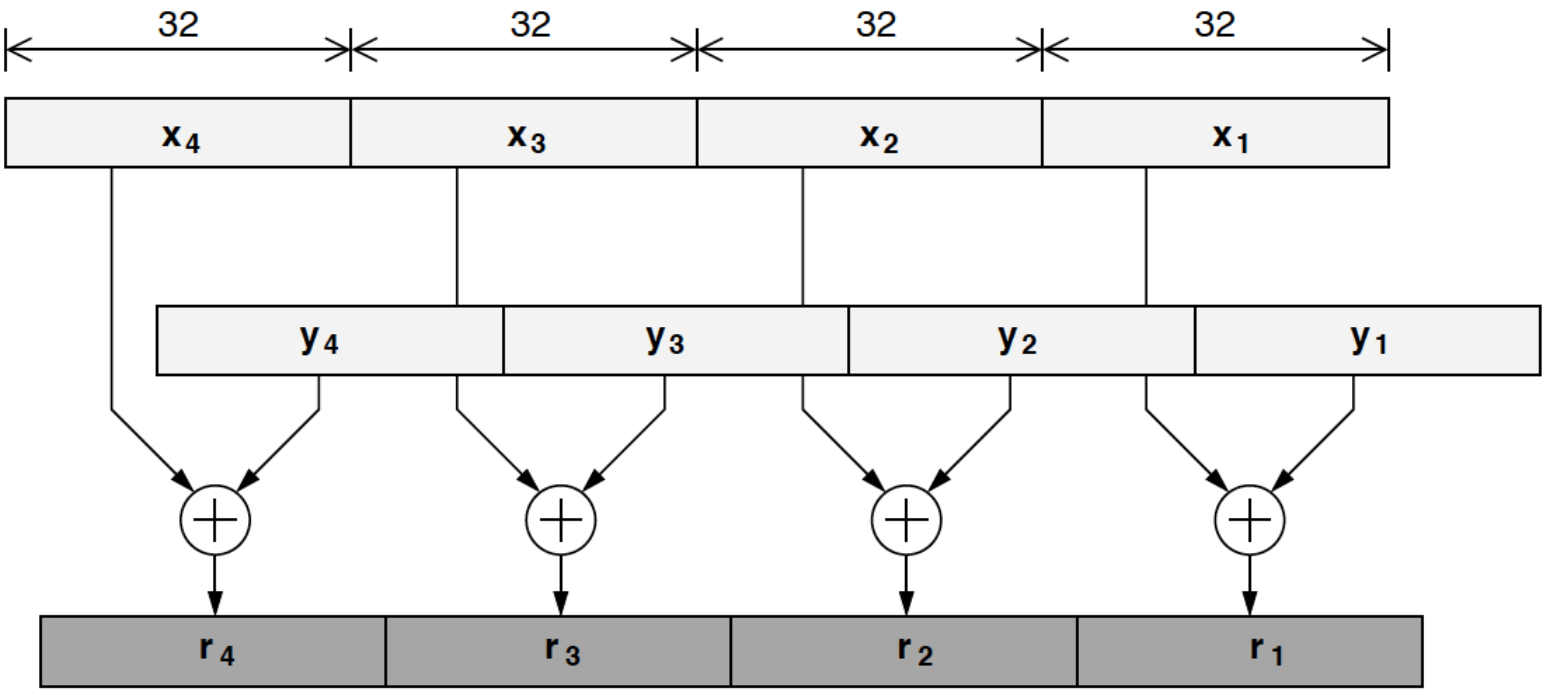


Figure 1.8: Example for SIMD: Single precision FP addition of two SIMD registers (x,y), each having a length of 128 bits. Four SP flops are executed in a single instruction.

Memory hierarchy

Data can be stored in a computer system in many different ways.

CPU has a set of registers, which can be accessed without delay.

In addition, there are several levels of *cache*, holding copies of recently used data items.

Main memory of a computer is much slower (than the caches).

Depiction of memory hierarchy

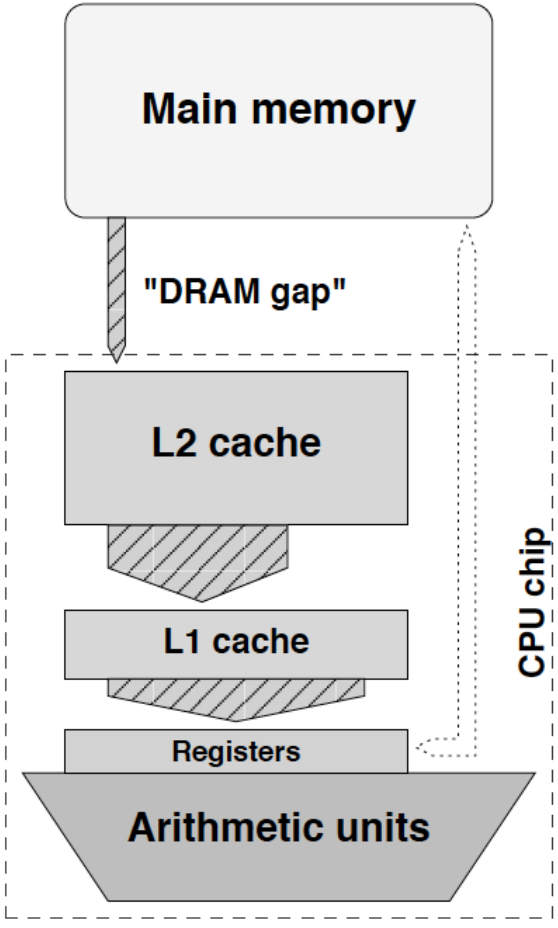
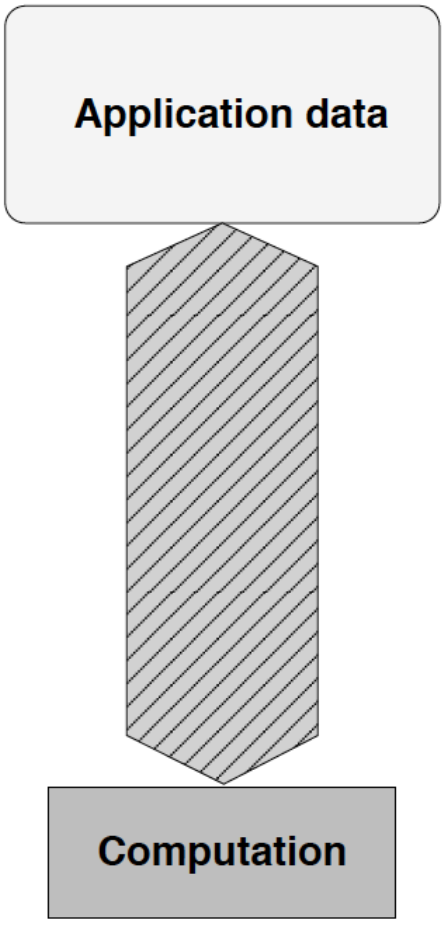


Figure 1.3: (Left) Simplified data-centric memory hierarchy in a cache-based microprocessor (direct access paths from registers to memory are not available on all architectures). There is usually a separate L1 cache for instructions. (Right) The "DRAM gap" denotes the large discrepancy between main memory and cache bandwidths. This model must be mapped to the data access requirements of an application.



Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die.

- L1 (level 1) data cache
- L1 instruction cache
- L2 and L3 (data & instruction) unified caches

The purpose of cache—reducing the impact of main memory's small bandwidth and high latency.

Cache hit and miss

Whenever the CPU issues a read request ("load") for transferring a data item to a register, the L1 data cache is checked. If the wanted data item is found in L1, this is called a *cache hit*, otherwise a *cache miss* occurs.

In case of a cache miss in L1, data must be fetched from upper cache levels or, in the worst case, from main memory.

Temporal locality

If data items loaded into a cache are to be used again "soon enough", then this is called good *temporal locality*.

Suppose accessing a data item in cache is a factor of τ faster than accessing the main memory. Let β denote the cache reuse ratio. Suppose access time to main memory is denoted by T_{m}, access time to cache thus $T_{\mathrm{c}} = T_{\mathrm{m}}/\tau$.

The average access time will be

$$T_{\mathrm{av}} = \beta T_{\mathrm{c}} + (1 - \beta) T_{\mathrm{m}}$$

Cache lines

The content of a cache is organized as *cache lines*. (A cache line has space for multiple data items.) This is for reducing the latency penalty for *streaming*—large amounts of data are loaded into the CPU, modified, and written back without the potential of reuse "in time".

All data transfers between caches and main memory happen on the cache line level.

If a code has good *spatial locality*, that is, the probability of successive accesses to neighboring items is high, the latency problem can be significantly reduced.

Cache eviction

If a data item needs to be loaded into a cache where all cache entries are occupied. One of the occupant entries has to be *evicted* by a hardware-implemented algorithm (typically following the *least-recently used* strategy) to give space.

“Common sense” optimizations

“Common sense” optimizations can be important

Very simple code changes can sometimes lead to significant performance boost.

The most important “common sense” principle: **avoiding performance pitfalls**!

Do less work; example 1

Example: assume `A` is an array of numerical values, and a prescribed threshold value: `threshold_value`.

```
int flag = 0;
for (i=0; i<N; i++) {
  if ( some_function(A[i]) < threshold_value )
    flag = 1;
}
```

Improvement: leave the loop as soon as `flag` becomes `1`.

```
int flag = 0;
for (i=0; i<N; i++) {
  if ( some_function(A[i]) < threshold_value ) {
    flag = 1;
    break;
  }
}
```

Do less work; example 2

```
for (i=0; i<500; i++)
  for (j=0; j<80; j++)
    for (k=0; k<4; k++)
      a[i][j][k] = a[i][j][k] + b[i][j][k]*c[i][j][k];
```

How many times is the `k`-indexed loop executed? And how many times for the `j`-indexed loop?

Do less work; example 2 (cont'd)

If the 3D arrays `a`, `b` and `c` have **contiguous** memory storage for all their values, then we can re-code as follows:

```
double *a_ptr = a[0][0];
double *b_ptr = b[0][0];
double *c_ptr = c[0][0];

for (i=0; i<(500*80*4); i++)
  a_ptr[i] = a_ptr[i] + b_ptr[i]*c_ptr[i];
```

This technique is called *loop collapsing*. The main motivation is to reduce loop overhead, may also help other (compiler-supported) optimizations.

Do less work; example 3

```
for (i=0; i<ARRAY_SIZE; i++) {
  a[i] = 0.;
  for (j=0; j<ARRAY_SIZE; j++)
    a[i] = a[i] + b[j]*d[j]*c[i];
}
```

Observation: `c[i]` is independent of the `j`-indexed loop.

Do less work; example 3 (cont'd)

Improvement:

```
for (i=0; i<ARRAY_SIZE; i++) {
  a[i] = 0.;
  for (j=0; j<ARRAY_SIZE; j++)
    a[i] = a[i] + b[j]*d[j];
  a[i] = a[i]*c[i];
}
```

Can we improve further?

Do less work; example 3 (further simplification)

There is a common factor:
`b[0]*d[0]+b[1]*d[1]+...+b[ARRAY_SIZE-1]*d[ARRAY_SIZE-1]`
which is unnecessarily re-computed in every `i` iteration!

```
t = 0.;
for (j=0; j<ARRAY_SIZE; j++)
  t = t + b[j]*d[j];

for (i=0; i<ARRAY_SIZE; i++)
  a[i] = t*c[i];
```

This technique is called *loop factoring* or *elimination of common subexpressions*.

Avoid expensive operations!

Special math functions (such as trigonometric, exponential and logarithmic functions) are usually very costly to compute.

An example from simulating non-equilibrium spins:

```
for (i=1; i<Nx-1; i++)
  for (j=1; j<Ny-1; j++)
     for (k=1; k<Nz-1; k++) {
      iL = spin_orientation[i-1][j][k];
      iR = spin_orientation[i+1][j][k];
      iS = spin_orientation[i][j-1][k];
      iN = spin_orientation[i][j+1][k];
      iO = spin_orientation[i][j][k-1];
      iU = spin_orientation[i][j][k+1];
      edelz = iL+iR+iS+iN+iO+iU;
      body_force[i][j][k] = 0.5*(1.0+tanh(edelz/tt));
    }
```

Example continued

If the values of `iL, iR, iS, iN, iO, iU` can only be -1 or $+1$, then the value of `edelz` (which is the sum of `iL, iR, iS, iN, iO, iU`) can only be $-6, -4, -2, 0, 2, 4, 6$.

If `tt` is a constant, then we can create a lookup table:

```
double tanh_table[13];
for (i=0; i<=12; i+=2)
  tanh_table[i] = 0.5*(1.0+tanh((i-6)/tt));
```



```
for (i=1; i<Nx-1; i++)
  for (j=1; j<Ny-1; j++)
     for (k=1; k<Nz-1; k++) {
       ....
       edelz = iL+iR+iS+iN+iO+iU;
       body_force[i][j][k] = tanh_table[edelz+6];
     }
```

Strength reduction

```
for (i=0; i<N; i++)
  y[i] = pow(x[i],3)/s;
```



```
double inverse_s = 1.0/s;
for (i=0; i<N; i++)
  y[i] = x[i]*x[i]*x[i]*inverse_s;
```

Shrinking the work set!

The *work set* of a code is the amount of memory it uses (or touches), also called *memory footprint*.

In general, shrinking the work set (if possible) is a good thing for performance, because it raises the probability of cache hit.

One example: The `spin_orientation` array should store values of type `char` instead of type `int`. (A factor of 4 in the difference of memory footprint.)

Avoiding branches

"Tight" loops: few operations per iteration, typically optimized by compiler using some form of pipelining. In case of conditional branches in the loop body, the compiler optimization will easily fail.

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++) {
    if (i>j)
      sign = 1.0;
    else if (i<j)
      sign = -1.0;
    else
      sign = 0.0;

    C[j] = C[j] + sign * A[j][i] * B[i];
  }
```

Avoiding branches (cont'd)

```
for (j=0; j<N-1; j++)
  for (i=j+1; i<N; i++)
    C[j] = C[j] + A[j][i] * B[i];

for (j=1; j<N; j++)
  for (i=0; i<j; i++)
    C[j] = C[j] - A[j][i] * B[i];
  }
```

We have got rid of the if-tests completely!

Another example of avoiding branches

```
for (i=0; i<n; i++) {
  if (i==0)
    a[i] = b[i+1]-b[i];
  else if (i==n-1)
    a[i] = b[i]-b[i-1];
  else
    a[i] = b[i+1]-b[i-1];
}
```

Another example of avoid branches (cont'd)

Using the technique of *loop peeling*, we can re-code as follows:

```
a[0] = b[1]-b[0];
for (i=1; i<n-1; i++)
  a[i] = b[i+1]-b[i-1];
a[n-1] = b[n-1]-b[n-2];
```

The role of compilers

A compiler translates a program, which is implemented in a programming language, to machine code.

A compiler can carry out code optimization of various degrees, dictated by the compiler options provided by the user. (`-O0`, `-O1`, `-O2`, ...)

Different compilers probably allow different compiler options, should refer to the user manual!

Numerical accuracy **may** suffer from too aggressive compiler optimizations.

Data access optimization

Importance of data access

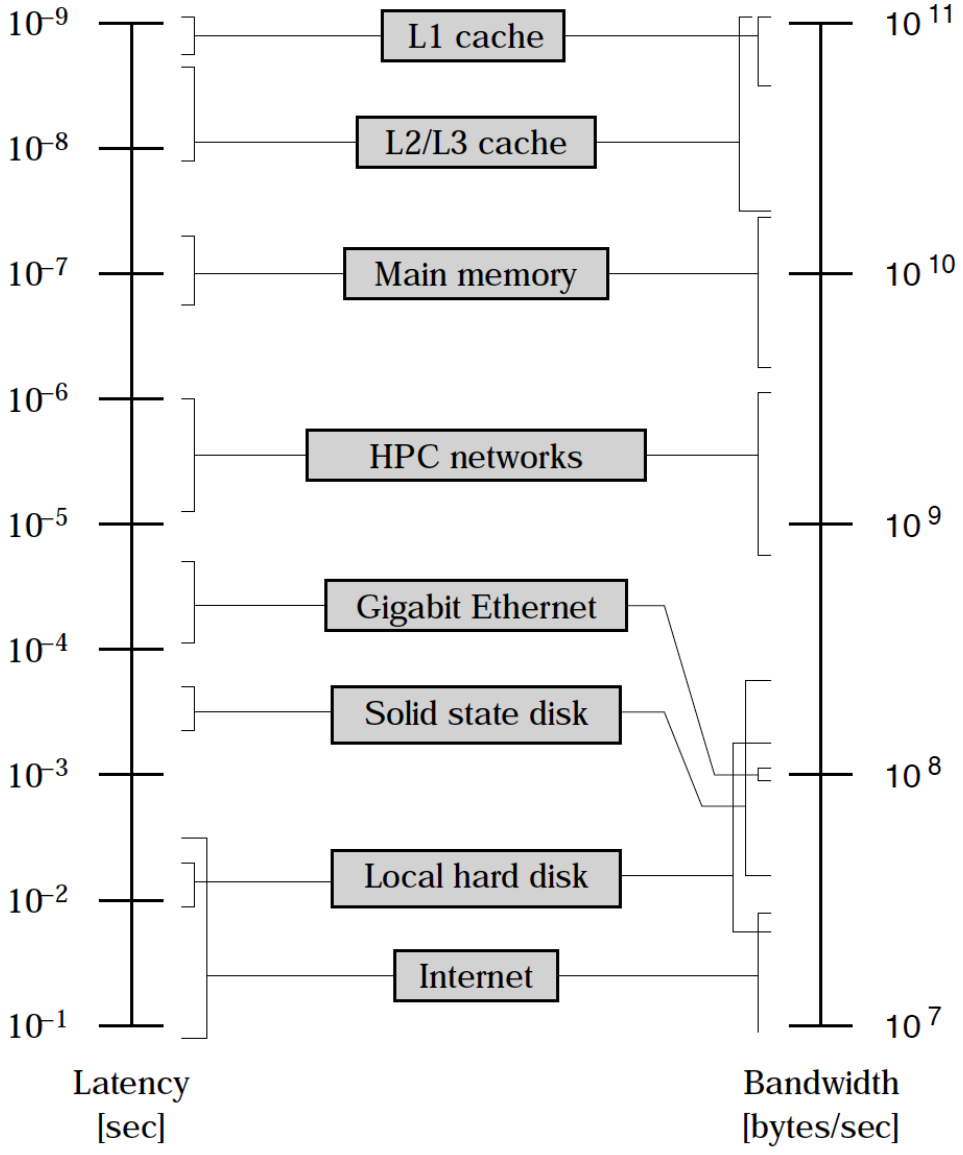
Applications in science and engineering mostly consist of **loop-based** code that moves large amounts of data in and out of the CPU.

Accessing data in the memory hierarchy (from L1 cache to main memory) is often the most prominent performance limiter.

Modern microprocessors have a very impressive theoretical peak performance (in number of FP operations *maximumly* executable per second), but the memory system is "**too slow**".

Typical latency and bandwidth numbers

$$\text{time usage} = \text{latency} + \frac{\text{data volume}}{\text{bandwidth}}$$



Rule of the thumb

Any optimization attempt, with respect to data access, should first aim at reducing traffic over slow data paths, or, making the data transfer as efficient as possible.

Understanding the “limitation”

Bandwidth-based performance modeling—to get a rough idea about the maximum performance for a code.

One can *estimate* the theoretically achievable performance of loop-based code, if it is bound by bandwidth limitations.

The concept of "machine balance"

Machine balance, B_{m}, of a processor is the ratio between the maximum memory bandwidth and the peak FP performance:

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

Access latency is assumed to be hidden completely (for example thanks to prefetch).

"Word" = one DP value (8 bytes)

"Memory bandwidth" could also be substituted by the bandwidth to caches or even network bandwidth.

The concept of "code balance"

To characterize a loop, we can calculate the **code balance** B_c:

$$B_\mathrm{c} = \frac{\text{data traffic [Words]}}{\text{floating-point operations [Flops]}}$$

That is, you should count the number of FP operations (easy), and also count (or estimate) the amount of data transfered over the performance-limiting data path (can be difficult).

Note: $\frac{1}{B_\mathrm{c}}$ is called **computational intensity**.

What is the expected maximum performance of a loop?

When you know the machine balance B_m of a CPU, and you want to run a loop that has B_c as its code balance.

What will be the maximum achievable performance P (in Flops/sec)?

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

Recall: P_max denotes the maximum FP performance, b_max denotes the maximum bandwidth of the performance-limiting data path.

Comparing P with P_{max}

- In case $P \approx P_{\text{max}}$: the achievable performance is not limited by bandwidth (so data access optimization is **not** needed).
- In case $P \ll P_{\text{max}}$: more analysis is needed to find out whether the code balance B_{c} can be improved, that is, making B_{c} smaller by data access optimization. (Note: smaller $B_{\text{c}} \rightarrow$ higher $P = \frac{b_{\text{max}}}{B_{\text{c}}}$)

Example of "balance analysis"

```
for (i=0; i<N; i++)
  A[i] = B[i] + C[i]*D[i];
```

- Each iteration has three loads (`B[i]`,`C[i]`,`D[i]`), one store (`A[i]`) and two floating-point operations
- Code balance: $B_\mathrm{c} = \frac{3+1}{2} = 2$
- If a CPU has machine balance $B_\mathrm{m} = 0.1$, then the maximumly achievable performance is $\frac{B_\mathrm{m}}{B_\mathrm{c}} P_\mathrm{max}$, that is, 5% of the peak FP performance
- On cache-based microprocessors, each store miss may incur a *cache line write allocate*, if non-temporal stores are not used. In that case, each store of `A[i]` in effect must be counted as a load plus a store, B_c thus becomes 2.5 $\rightarrow$ only 4% of P_max is maximumly achievable.

Read about "non-temporal stores" at `https://vgatherps.github.io/2018-09-02-nontemporal/`

How realistic is b_{max}?

In reality, even the simplest memory-intensive loops are not able to achieve the theoretical hardware maximum memory bandwidth b_{max}.

The well-known stream micro-benchmarks can be used to measure the realistically achievable maximum memory bandwidth.

STREAM micro-benchmarks

Four micro-benchmarks (https://www.cs.virginia.edu/stream/)

type	kernel	DP words	flops	B_c
COPY	`A(:)=B(:)`	2 (3)	0	N/A
SCALE	`A(:)=s*B(:)`	2 (3)	1	2.0 (3.0)
ADD	`A(:)=B(:)+C(:)`	3 (4)	1	3.0 (4.0)
TRIAD	`A(:)=B(:)+s*C(:)`	3 (4)	2	1.5 (2.0)

Table 3.2: The STREAM benchmark kernels with their respective data transfer volumes (third column) and floating-point operations (fourth column) per iteration. Numbers in brackets take write allocates into account.

More realistic "balance analysis"

We will from now on use the realistically achievable memory bandwidth, b_{S}, which is measured by STREAM.

Then, the realistically achievable maximum FP performance is estimated as

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

Use unit-stride to access arrays, if possible

Assume that 2D array `A` has row-major storage order.

```
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    A[i][j] = i*j;   // stride-1 access, good

for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    A[j][i] = i*j;   // stride-N access, bad!!!
```

Access optimization for algorithm class $O(N)/O(N)$

Algorithm class $O(N)/O(N)$

- 1D loops (N: loop length)
- 1D arrays (N: array length)

Normally not much room for data access optimization, but *loop fusion* can **sometimes** help.

Example of loop fusion

Original code: two loops after each other:

```
for (i=0; i<N; i++) {
  A[i] = B[i] + C[i];
}

for (i=0; i<N; i++) {
  Z[i] = B[i] + E[i];
}
```

- Number of floating-point operations: $2N$
- Number of memory loads & stores: $4N + 2N$

Code balance: $B_c = \frac{6}{2}$, can we improve?

Example of loop fusion (cont'd)

Loop fusion:

```
for (i=0; i<N; i++) {
  A[i] = B[i] + C[i];
  Z[i] = B[i] + E[i];
}
```

- Now each $B[i]$ value is only loaded once instead of twice!
- New code balance: $B_{\mathrm{c}} = \frac{5}{2}$
- Loop fusion will also reduce looping overhead
- Beware of the limited register resources: The code body of each iteration shouldn't be too large. (Otherwise, *register spilling* can lead to performance degradation.)

Access optimization for algorithm class $O(N^2)/O(N^2)$

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

- Two-level loop nests (N: loop length on each level)
- Number of floating-point operations: $O(N^2)$
- Number of memory loads & stores: $O(N^2)$

There is more room for data access optimization (than the class of $O(N)/O(N)$)

Example of data access optimization for $O(N^2)/O(N^2)$

Dense matrix-vector multiply

```
for (i=0; i<N; i++) {
  double tmp = C[i];
  for (j=0; j<N; j++)
    tmp += A[i][j]*B[j];
  C[i] = tmp;
}
```

- Number of FP: $2N^2$
- Number of loads & stores: N^2 for 2D array `A`, $2N$ for 1D array `C`
- But, how many loads are associated with 1D array `B`?
  - Small cache $\rightarrow$ array `B` is loaded N times $\rightarrow$ N^2 memory loads
  - Large cache $\rightarrow$ array `B` is loaded only once $\rightarrow$ N memory loads

Illustration of array B being loaded N times

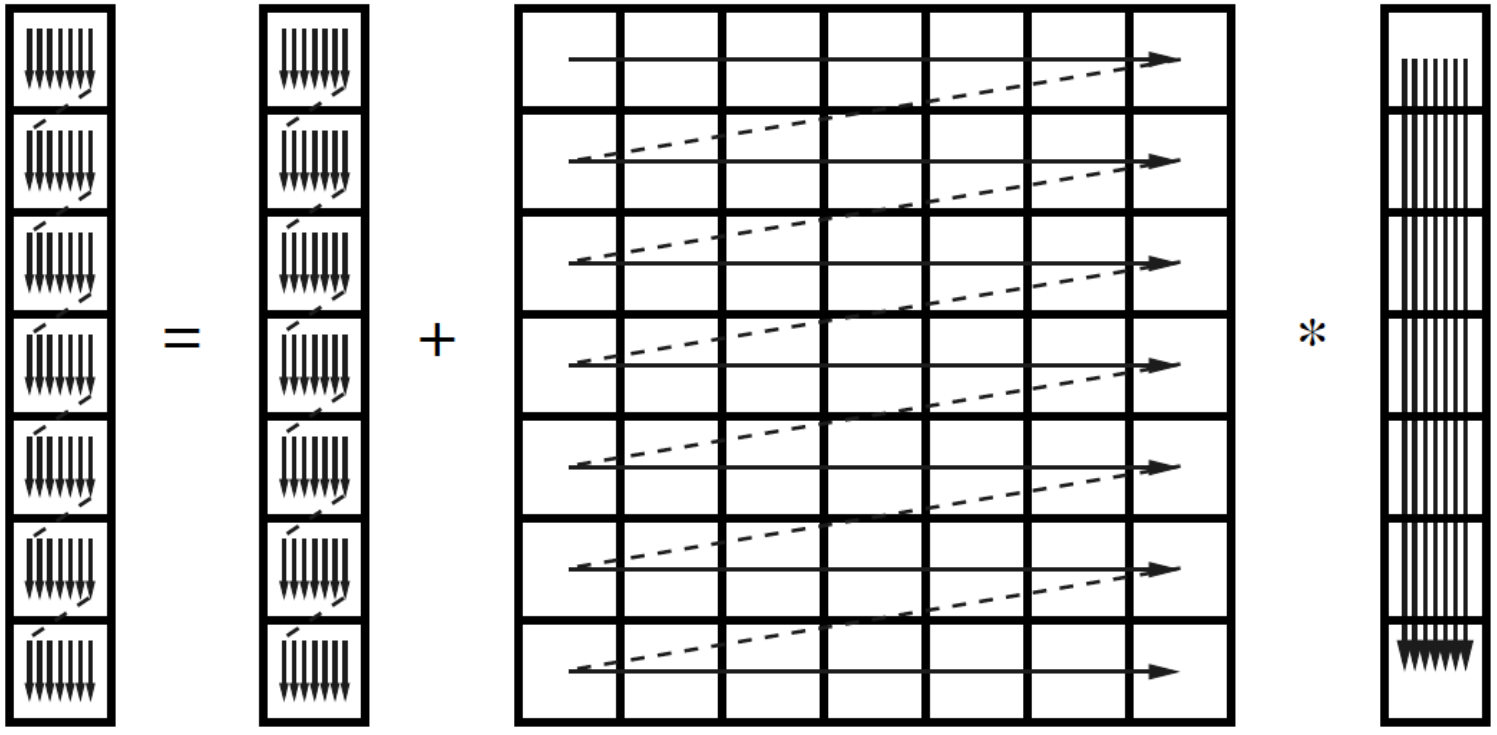


Figure 3.11: Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded N times.

Loop *unrolling*

m-way unroll and jam:

```
for (i=0; i<N; i+=m) {
  for (j=0; j<N; j++) {
    C[i+0] += A[i+0][j]*B[j];
    C[i+1] += A[i+1][j]*B[j];
    // ...
    C[i+m-1] += A[i+m-1][j]*B[j];
  }
}
// remainder code in case (N%m)>0 ....
```

- *m*-fold reuse of each `B[j]` from register
- Total number of memory loads and stores: $N^2 + N^2/m + 2N$ (for small cache size)
- Size of *m* shouldn't be too large, to avoid too high *register pressure*

Illustration of the effect of unrolling

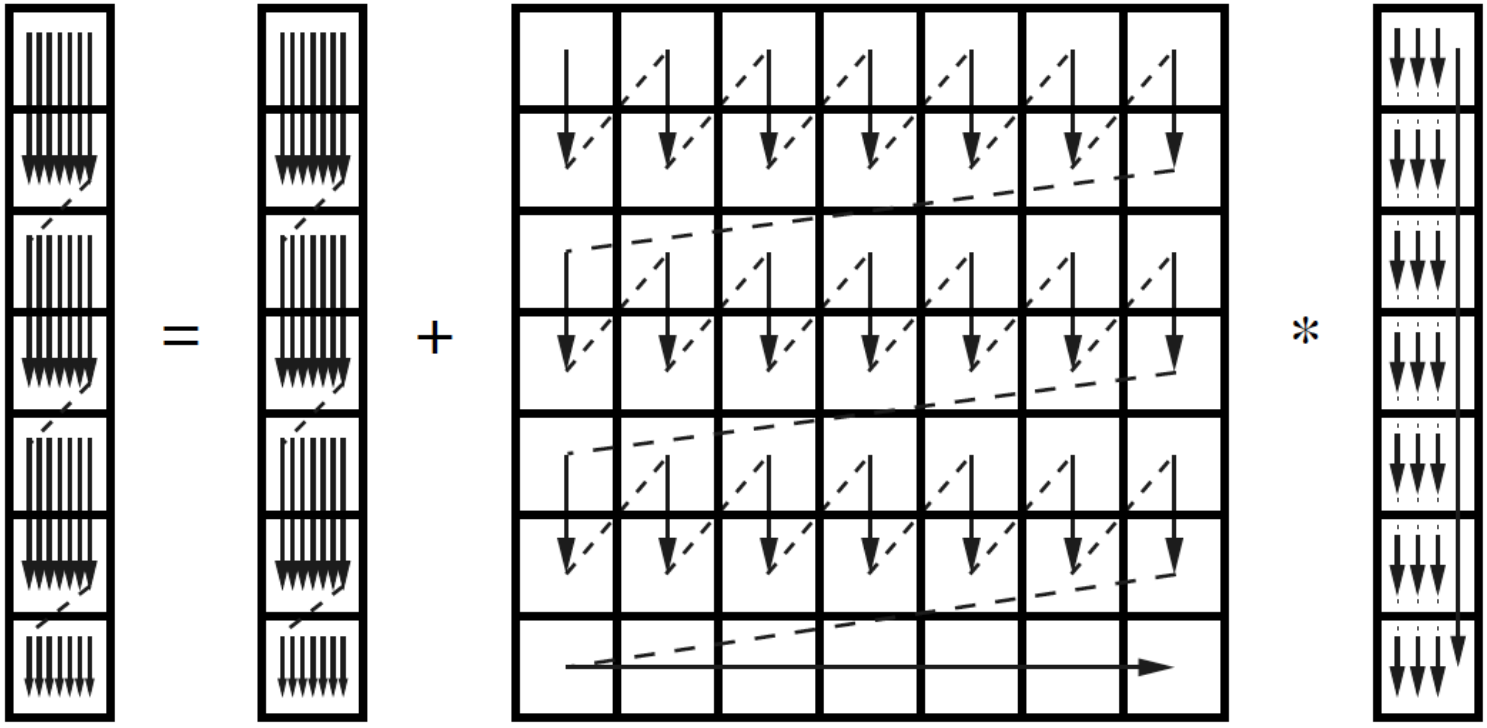


Figure 3.12: Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

Another $O(N^2)/O(N^2)$ algorithm: matrix transpose

$$A = B^T$$

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    A[j][i] = B[i][j];
```

Both `A` and `B` are assumed to be 2D arrays with row-major storage.

Very large jumps in memory associated with loading `B[i][j]` $\rightarrow$ very bad cache line utilization.

Loop unrolling applied to matrix transpose

```
for (j=0; j<N; j+=m)
  for (i=0; i<N; i++) {
    A[j+0][i] = B[i][j+0];
    A[j+1][i] = B[i][j+1];
    // ....
    A[j+m-1][i] = B[i][j+m-1];
  }
```

Illustration

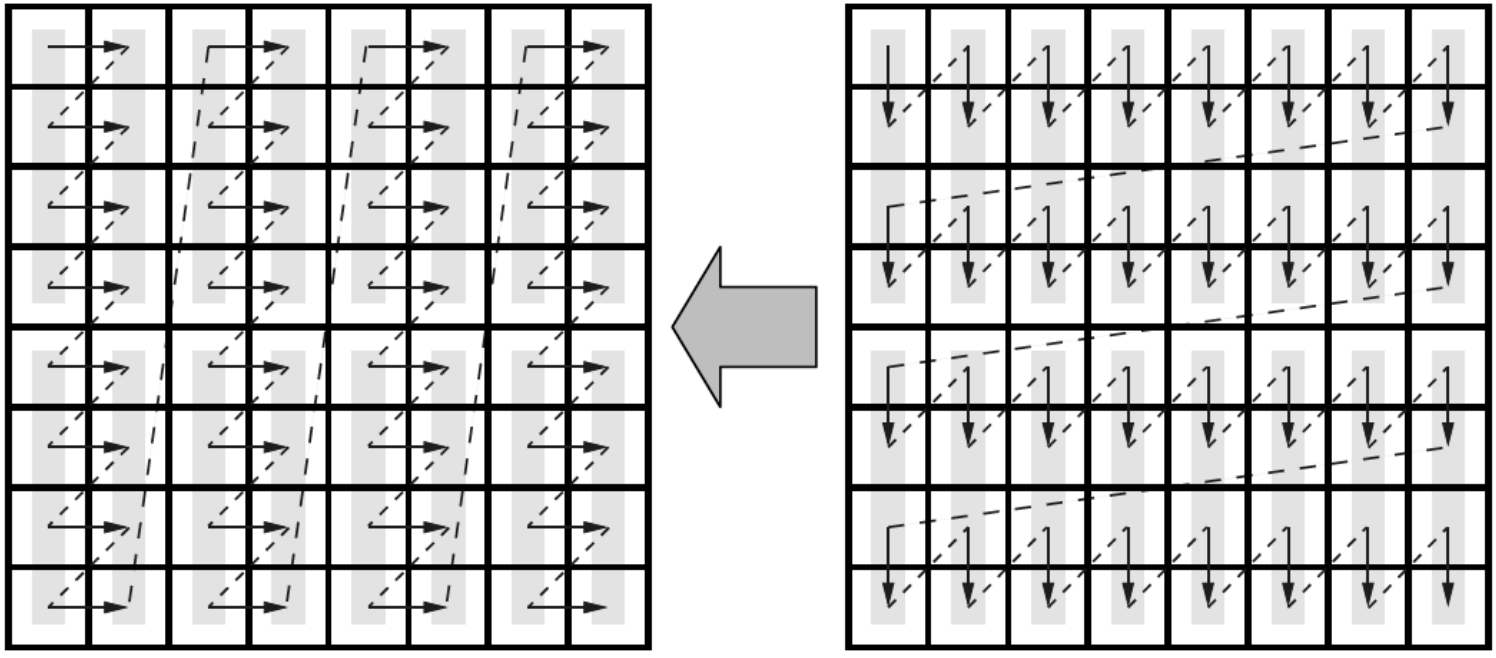


Figure 3.13: Two-way unrolled "flipped" matrix transpose (i.e., with strided load in the original version).

B *A*

Loop blocking + unrolling

```
for (jj=0; jj<N; jj+=b) {
  jstart = jj; jstop = jj+b-1;
  for (ii=0; ii<N; ii+=b) {
    istart = ii; istop = ii+b-1;

    for (j=jstart; j<=jstop; j+=m)
      for (i=istart; i<=istop; i++) {
        A[j+0][i] = B[i][j+0];
        A[j+1][i] = B[i][j+1];
        // ....
        A[j+m-1][i] = B[i][j+m-1];
      }
  }
}
```

Illustration

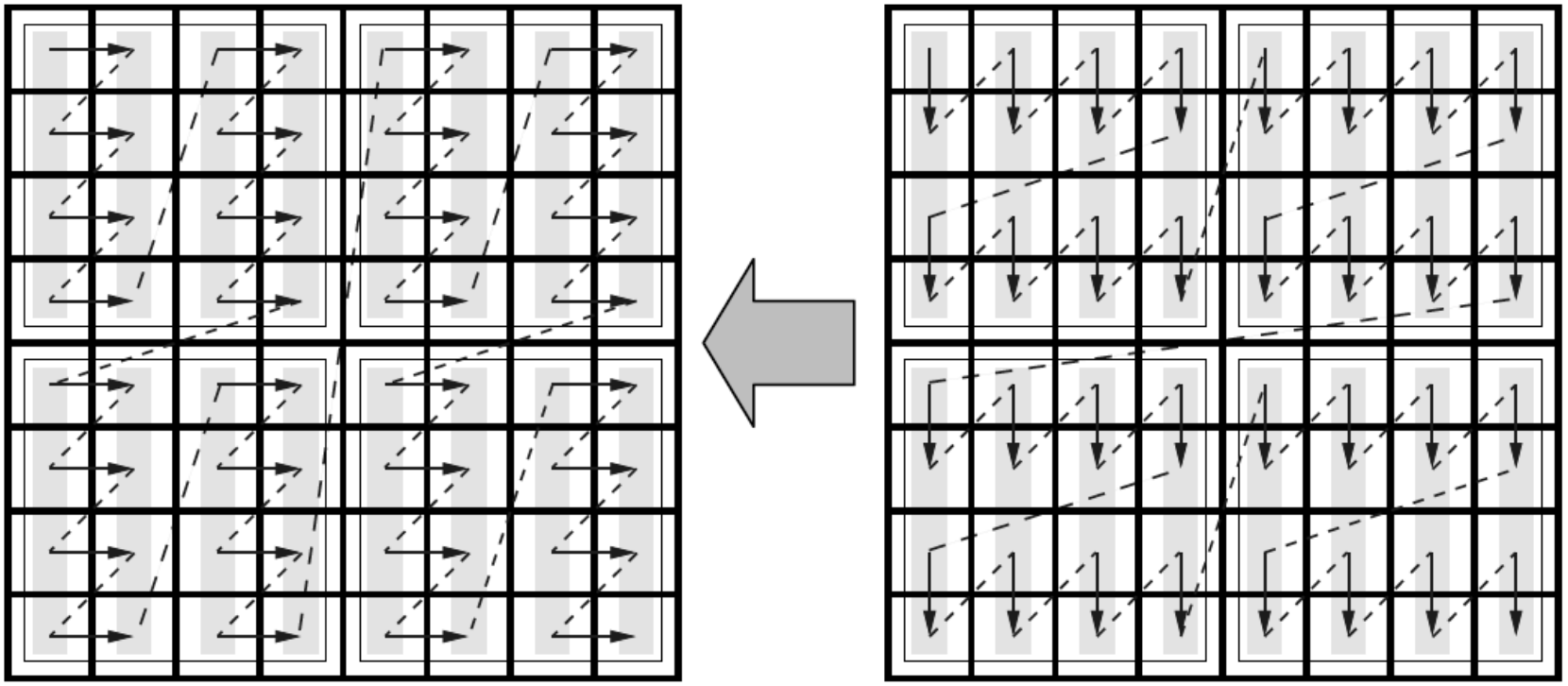


Figure 3.14: 4×4 blocked and two-way unrolled "flipped" matrix transpose.

B A

Access optimization for algorithm class $O(N^2)/O(N)$

Example:

```
double sum = 0.;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++)
    sum = sum + foo(A[i],B[j])
}
```

- Array `B` has the risk of being loaded N times (when N is very large)
- Total number of memory loads: $N + N^2$ (first part for array `A`, second part for array `B`)

Applying loop blocking

```
double sum = 0.;

for (jj=0; jj<N; jj+=b) {
  jstart = jj; jstop = jj+b-1;

  for (i=0; i<N; i++) {
    for (j=jstart; j<=jstop; j++)
      sum = sum + foo(A[i],B[j])
  }
}
```

- Appropriate choice of b will allow array B to be loaded from memory only once.
- Array A will now be loaded N/b times (instead of only once).
- Total number of memory loads: $N^2/b + N$

Exercises & a case study

Recap of cache lines

The content of a cache is organized as **cache lines**. (Each cache line has space for multiple data items.)

All data transfers between caches and main memory happen on the cache line level. (Must load/store an entire cache line, cannot only load/store a single data item.)

When a new cache line is loaded into the cache, but all its possible locations are occupied, one of the old occupant cache lines needs to be "kicked out" (evicted). The most commonly used policy is to evict the least-recently used cache line.

Exercise 1

Assume that we have a CPU with 40 GB/s as the theoretical memory bandwidth and 100 GFLOP/s (in double precision) as the theoretical peak floating-point performance. Can you estimate how much time the following code needs on this CPU, when $n = 10^{10}$?

```
double s = 0.0;
for (i=0; i<n; i++) {
   s += a[i]*a[i];
}
```

We assume that array `a` is of length n and contains double-precision values.

Exercise 2

Explain why the following code will probably run very slowly. How will you modify the code such that the same computation is done but the performance is considerably improved?

```
for (int i=0; i<n; i++) {
  c[i] = exp(1.0*i/n)+sin(3.1415926*i/n);
  for (int j=0; j<n; j++)
    a[j][i] = b[j][i] + d[j]*e[i];
}
```

We assume that both `a` and `b` are row-major 2D arrays of dimension $n \times n$, while `c`, `d`, `e` are 1D arrays of length n.

Case study: The 2D Jacobi algorithm

Let us look at the following computation:

```
for (it=0; it<itmax; it++) {
  for (k=1; k<kmax-1; k++)
    for (i=1; i<imax-1; i++)
      phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                       +phi[k][i+1]+phi[k+1][i])*0.25;
  /* pointer swapping */
  temp_ptr = phi_new;
  phi_new = phi;
  phi = temp_ptr;
}
```

Note: both `phi_new` and `phi` are 2D arrays (row-major storage)

2D Jacobi: performance prediction

Suppose a computer has only one level of cache, please consider the following two situations and derive the code balance for each situation:

- The cache is extremely small, that is, not enough to even store one row of `phi`.
- Suppose the cache can store at least two rows of `phi`, but not enough to store the entire array `phi`.

When you have derived the code balance, how would you predict the actual performance of the 2D Jacobi iterations, in terms of number of floating-point operations per second?