IN3200/IN4200: Chapter 3 Data access optimization (Part 1)

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

Objectives

- What is the maximumly achievable performance?
  - Balance analysis and “lightspeed” estimates
- Data access optimization techniques

Importance of data access

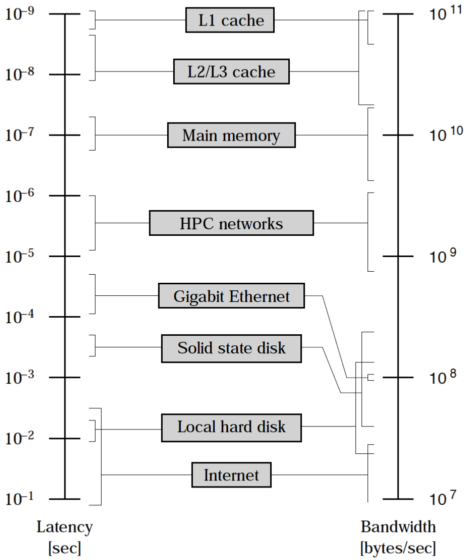
Applications in science and engineering mostly consist of **loop-based** code that moves large amounts of data in and out of the CPU.

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

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



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 memory/cache/network 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:

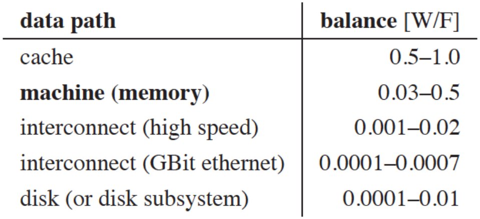
$$B_\mathrm{m} = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak FP performance [GFlops/sec]}} = \frac{b_\mathsf{max}}{P_\mathsf{max}}$$

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

“Word” = one double-precision (DP) value: **8 bytes**

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

Example values of machine balance



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

Table 3.1: 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.

The above values are somewhat outdated.

The increase of memory bandwidth typically falls behind the increase of FP performance—the ever-increasing **DRAM gap**.

A relatively recent example of machine balance

Intel Skylake Platinum 28-core CPU (model 8180) from year 2017:

- Peak memory bandwidth:
  6 memory channels $\times$ 2.666 GT/sec $\times$ 1 word/transfer $=$
  16 GWords/sec (G: giga (10^9) T: transfer)
- Peak double-precision FP performance:
  28 cores $\times$ 2.3 GHz AVX-512 clock rate $\times$ 32 Flops/cycle $=$
  2061 GFlops/sec (Each core: 2 FP pipelines, 512-bit SIMD, fuzed multiply-add)
- So the machine balance is only $\frac{16}{2061} = 0.00776$

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)?

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

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 $P \approx P_{\max}$: the achievable performance is not limited by bandwidth (so data access optimization is **not** needed).
- In case $P \ll P_{\max}$: 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 $B_c \rightarrow$ higher $P = \frac{b_{\max}}{B_c}$)

“Lightspeed” of a loop

$$\frac{P}{P_{\mathsf{max}}} = \mathsf{min}\left(1, \frac{B_{\mathrm{m}}}{B_{\mathrm{c}}}\right)$$

is the maximum achievable fraction of peak performance for a code with balance B_{c} on a machine with balance B_{m}—also called the **lightspeed** of a loop.

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: $B_c = \frac{3+1}{2} = 2$
- If a CPU has machine balance $B_m = 0.1$, then the maximumly achievable performance is $\frac{B_m}{B_c} P_{\max}$, 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/

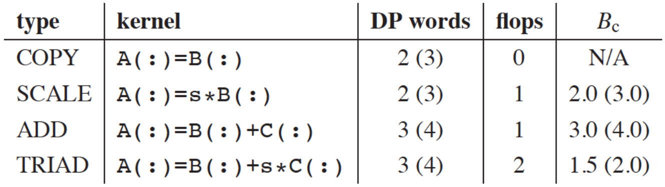
How realistic is $b_{\max}$?

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

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

Storage order of multi-dimensional arrays

Multi-dimensional arrays normally have an underlying contiguous 1D storage.

C program typically adopts a **row-major** storage order.

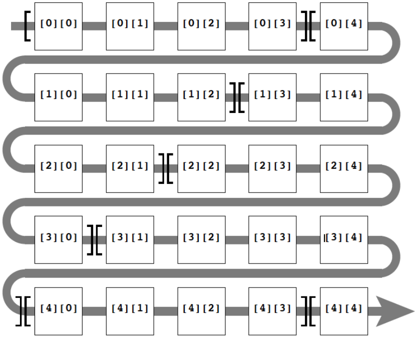


Figure 3.3: Row major order matrix storage scheme, as used by the C programming language. Matrix rows are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.

Storage order of multi-dimensional arrays (cont'd)

Fortran program typically adopts a **column-major** storage order.

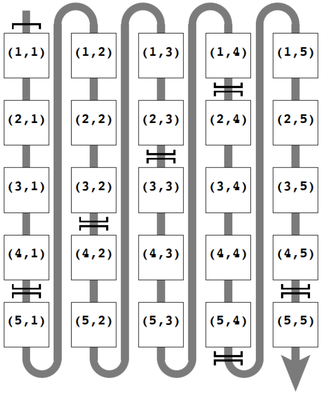


Figure 3.4: Column major order matrix storage scheme, as used by the Fortran programming language. Matrix columns are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.

(Read the textbook with care, because most coding examples are in Fortran.)

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

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

Cache lines (repetition of knowledge from Chapter 1)

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.

Case study: The 2D Jacobi algorithm

Skipping the mathematical and numerical details (which are given in the textbook), let us focus on 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]+phi[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

Balance analysis applied to 2D Jacobi:

- 4 floating-point operations per (k, i)
- 1 store to memory per (k, i)
- **How many loads from memory per** (k, i)**?**
  (*It depends on the cache size.*)

An important observation

During every `it` iteration, each `phi[k][i]` value (except those on the boundary) will participate in computing **4** different values of `phi_new`:

1. as the "below" neighbor when computing `phi_new[k-1][i]`
2. as the "right" neighbor when computing `phi_new[k][i-1]`
3. as the "left" neighbor when computing `phi_new[k][i+1]`
4. as the "above" neighbor when computing `phi_new[k+1][i]`

A question to ponder first

What will be the code balance, if there is no cache?