High performance computing

A. Scemama

21/11/2024

Laboratoire de Chimie et Physique Quantiques, Univ. Toulouse/CNRS

HPC architectures

Computers

In 2021 (order of magnitude)

- 1 socket (x86 CPU @ 2.2-3.3 GHz, 4 cores, hyperthreading)
- $\sim$ 4-16 GB RAM
- $\sim$ 500GB SSD
- Graphics card : ATI Radeon, Nvidia GeForce
- Gigabit ethernet
- USB, Webcam, Sound card, etc
- $\sim$ 500 euros

Computer designed for computation

In 2021 (order of magnitude)

- 2 sockets (x86 CPU @ 2.2 GHz, 32 cores/socket, hyperthreading)
- 64-128 GB RAM
- Multiple SSD HDDs (RAID0)
- Gigabit ethernet
- Possibly an Accelerator (Nvidia Volta/Ampere)
- $\sim$ 5k euros

Cluster

- Many computers designed for computation
- Compact (1-2U in rack) / machine
- Network switches
- Login server
- Batch queuing system (SLURM / PBS / SGE / LFS)
- Cheap Cooling system
- Requires a lot of electrical power ($\sim$ 10kW/rack)
- Possibly a Low-latency / High-bandwidth network (Infiniband or 10Gb ethernet)
- >50k euros

Supercomputer

- Many computers designed for computation
- Very compact (<1U in rack) / machine
- Low-latency / High-bandwidth network (Infiniband or 10Gb ethernet)
- Network switches
- Parallel filesystem for scratch space (Lustre / BeeGFS / GPFS)
- Multiple login servers
- Batch queuing system (SLURM / PBS / SGE / LFS)
- Highly efficient cooling system (water)
- Requires a lot of electrical power (>100kW)

Definitions

Top500	Rank of the 500 fastest supercomputers
Flop	Floating point operation
Flops	Flop/s, Number of Floating point operations per second
RPeak	Peak performance, max possible number of Flops
RMax	Real performance on the Linpack benchmark (dense eigenproblem)
SP	Single precision (32-bit floats)
DP	Double precision (64-bit floats)
FPU	Floating Point Unit
FMA	Fused multiply-add ($a \times x + b$ in 1 instruction)

Quantifying performance

Example

RPeak of Intel Xeon Gold 6140 Processor :

- 18 cores
- 2.3 GHz
- 2 FPUs
- 8 FMA (DP)/FPU/cycle

$18 \times 2.3\,10^9 \times 2 \times 8 \times 2 = 1.3$ TFlops (DP)

Number of hours 730/month, 8760/year

Units Kilo (K), Mega (M), Giga (G), Tera (T), Peta (P), Exa (E), ...

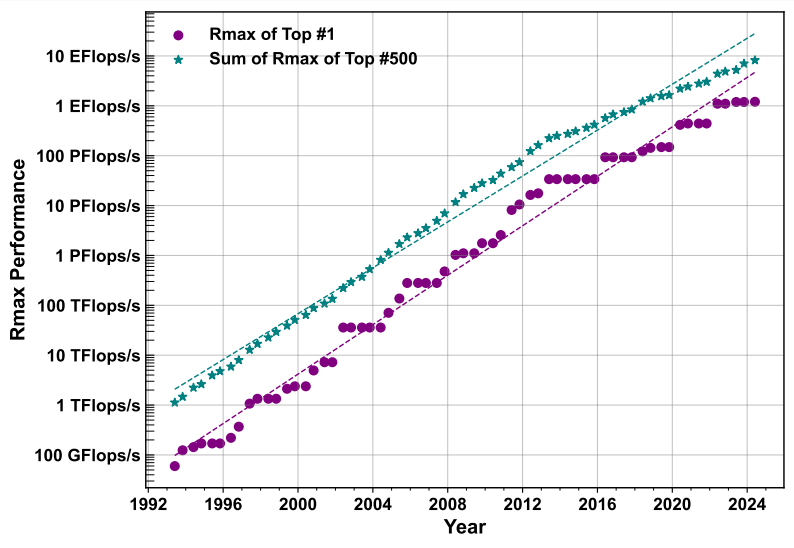
Top500 (1996)

Rank	System	Cores	Rmax (GFlop/s)	Rpeak (GFlop/s)	Power (kW)
1	CP-PACS/2048, Hitachi/Tsukuba Center for Computational Sciences, University of Tsukuba Japan	2,048	368.2	614.4	
2	Numerical Wind Tunnel, Fujitsu National Aerospace Laboratory of Japan Japan	167	229.0	281.3	498
3	SR2201/1024, Hitachi University of Tokyo Japan	1,024	220.4	307.2	
4	XP/S140, Intel Sandia National Laboratories United States	3,680	143.4	184.0	
5	XP/S-MP 150, Intel DOE/SC/Oak Ridge National Laboratory United States	3,072	127.1	154.0	

Top500 (2024)

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

Performance over years



Curie thin nodes (TGCC, France)

Ranked 9th in 2012, 77 184 cores, 1.7 PFlops, 2.1 MW

Mare Nostrum (BSC, Spain)

Ranked 13th in 2017, 153 216 cores, 6.5 PFlops, 1.6 MW

Architecture

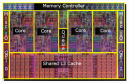
Supercomputer

Islands

Nodes

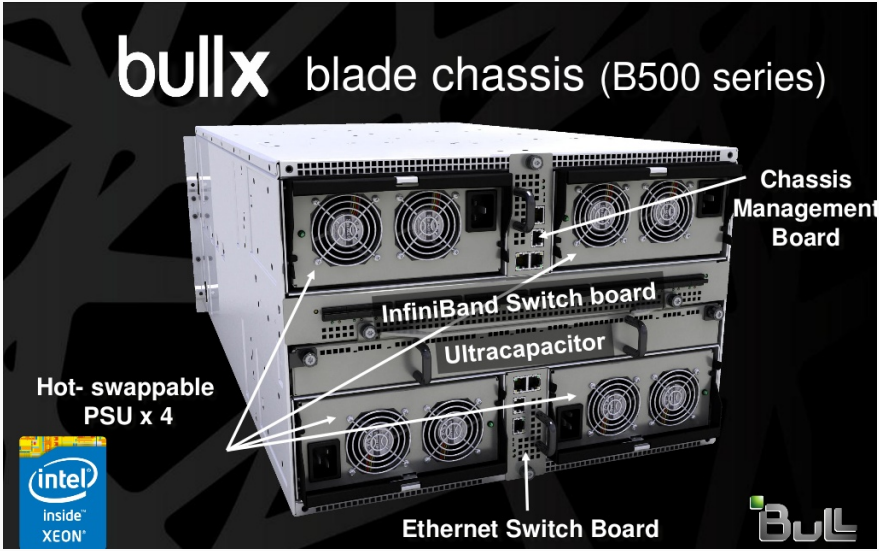
Sockets

Cores

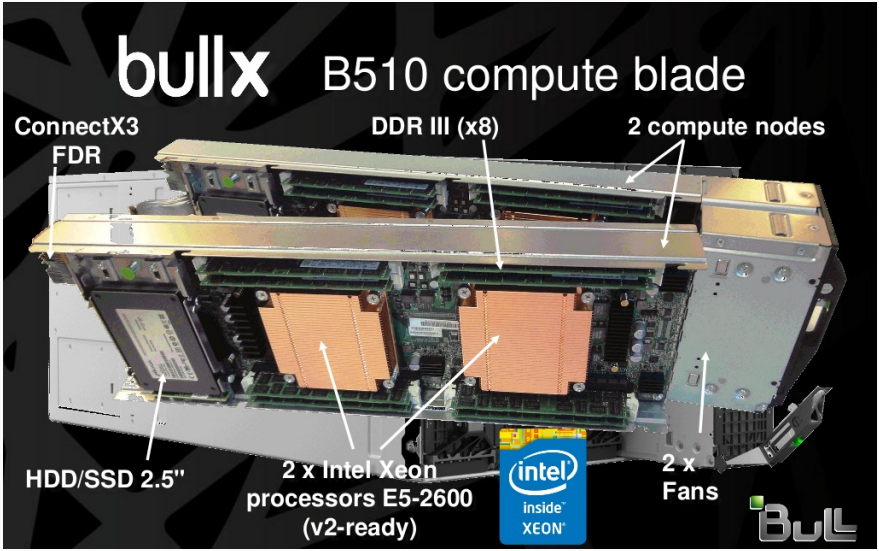


Chassis (Front)

Chassis (Back)



Compute Node



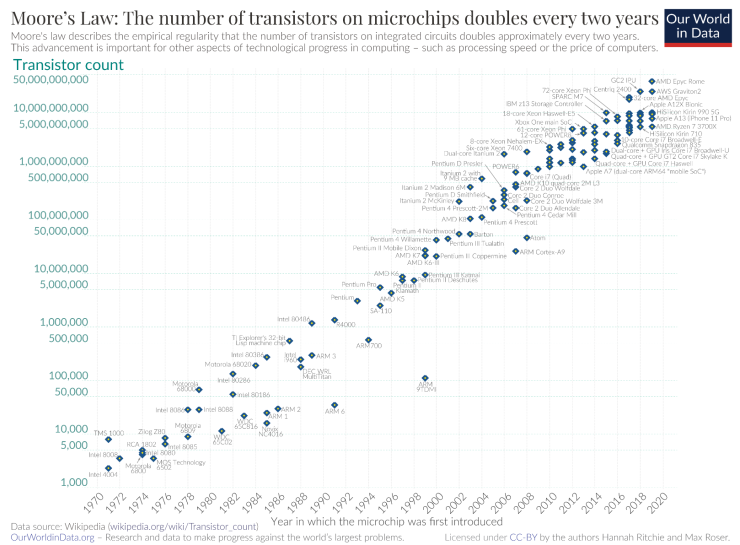
Socket

Why such an architecture?

Moore's "Law"

- *The number of transistors doubles every two years.*
- Often interpreted as *the computational power doubles every two years.*
- Exponential law $\Longrightarrow$ will fail.
- 4 nm semiconductors in 2024: the atomic limit is approaching.

Moore’s Law



Moore’s Law: The number of transistors on microchips doubles every two years
Our World in Data
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.
Transistor count
50,000,000,000
10,000,000,000
5,000,000,000
1,000,000,000
500,000,000
100,000,000
50,000,000
10,000,000
5,000,000
1,000,000
500,000
100,000
50,000
10,000
5,000
1,000
1970 1972 1974 1976 1978 1980 1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018 2020
Year in which the microchip was first introduced
GC2 IPU
AMD Epyc Rome
72-core Xeon Phi
Centriq 2400
AWS Graviton2
SPARC M7
32-core AMD Epyc
IBM z13 Storage Controller
Apple A12X Bionic
18-core Xeon Haswell-E5
HiSilicon Kirin 990 5G
Apple A13 (iPhone 11 Pro)
Xbox One main SoC
AMD Ryzen 7 3700X
61-core Xeon Phi
HiSilicon Kirin 710
12-core POWER8
8-core Xeon Nehalem-EX
10-core Core i7 Broadwell-E
Six-core Xeon 7400
Qualcomm Snapdragon 835
Dual-core Itanium 2
Dual-core + GPU Iris Core i7 Broadwell-U
Quad-core + GPU GT2 Core i7 Skylake K
Pentium D Presler
POWER6
Quad-core + GPU Core i7 Haswell
Apple A7 (dual-core ARM64 "mobile SoC")
Itanium 2 with 9 MB cache
Core i7 (Quad)
AMD K10 quad-core 2M L3
Itanium 2 Madison 6M
Core 2 Duo Wolfdale
Pentium D Smithfield
Core 2 Duo Conroe
Itanium 2 McKinley
Cell
Core 2 Duo Wolfdale 3M
Pentium 4 Prescott-2M
Core 2 Duo Allendale
Pentium 4 Cedar Mill
AMD K8
Pentium 4 Prescott
Pentium 4 Northwood
Barton
Atom
Pentium 4 Willamette
Pentium III Tualatin
Pentium II Mobile Dixon
ARM Cortex-A9
AMD K7
Pentium III Coppermine
AMD K6-III
AMD K6
Pentium III Katmai
Pentium II Deschutes
Pentium Pro
Pentium II Klamath
Pentium
AMD K5
SA-110
Intel 80486
R4000
TI Explorer's 32-bit Lisp machine chip
ARM700
Intel 80386
Intel i960
ARM 3
Motorola 68020
DEC WRL MultiTitan
Intel 80286
ARM 9TDMI
Motorola 68000
Intel 80186
Intel 8086
Intel 8088
ARM 2
ARM 1
ARM 6
WDC 65C816
Novix NC4016
Motorola 6809
WDC 65C02
TMS 1000
Zilog Z80
RCA 1802
Intel 8085
Intel 8008
Intel 8080
Motorola 6800
MOS Technology 6502
Intel 4004
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world’s largest problems.
Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

Why such an architecture?

Heat dissipation of a processor: $D = F \times V^2 \times C$

- F : Frequency ($\sim$ 3 GHz)
- V : Electrical voltage
- C : Constant related to the size of semiconductors (nm)

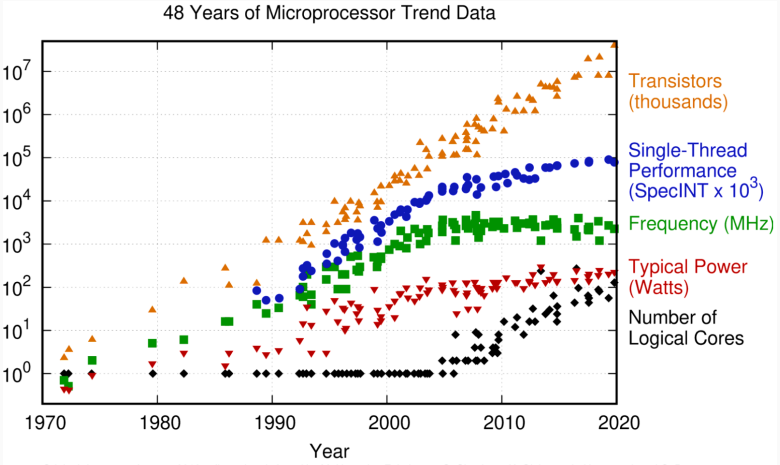
To keep the processor stable, V needs to be increased linearly with the frequency, so $D \propto F^3$.

Goal: Make $2\times$ more flops/s

- Double frequency: $\rightarrow D \times 8$
- Use 2 CPUs: $\rightarrow D \times 2$
- Use accelerators (GPU, FPGA, ...): $\rightarrow D + D'$

Taking advantage of modern hardware $\Longrightarrow$ Parallelism

Evolution over years



Evolution of CPUs in terms of number of transistors, power consumption and number of cores.
From Karl Rupp Github repo Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten" at 1970,6e-3 tc
ls 1 font ",8" New plot and data collected for 2010-2019 by K. Rupp

Parallelism is everywhere

Parallelism is not only MPI/OpenMP:

- Multiple compute nodes (MPI)
- Multiple processors/nodes (MPI/OpenMP)
- Multiple cores/processor (MPI/OpenMP)
- Multiple floating point (FP) units / core (Compiler)
- Single Instruction Multiple Data (SIMD) Vectorization (Compiler)
- Out of order execution (CPU)
- GPU accelerators
- Multiple memory channels
- Multiple network interfaces
- Parallel file systems
- *etc*

Fundamentals of parallelization

Definitions

Concurrency Running multiple computations at the same time.

Parallelism Running multiple computations on different execution units.

Multiple levels

Distributed	Multiple machines
Shared memory	Single machine, multiple CPU cores
Hybrid	With accelerators (GPUs, ...)
Instruction-level	Superscalar processors
Bit-level	Vectorization

All levels of parallelism can be exploited in the same code, but every problem is not parallelizable at all levels.

Amdahl's law

Definition

If P is the proportion of a program that can be made parallel, the maximum speedup that can be achieved is:

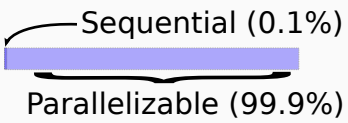
$$S_{\text{max}} = \frac{1}{(1-P) + P/n}$$

S : Speedup

P : Proportion of a program that can be made parallel

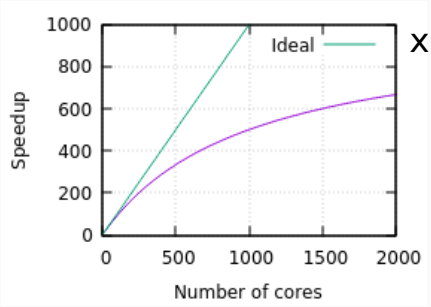
n : Number of cores

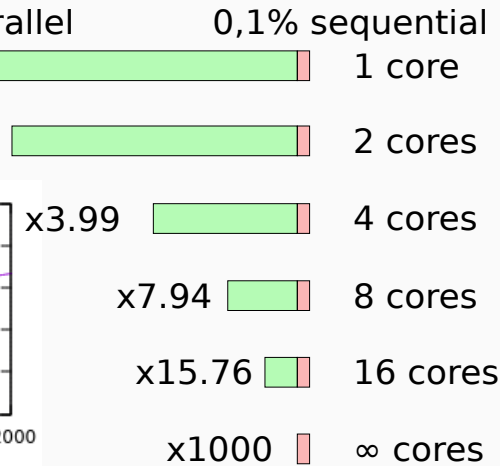
Example



- Max speedup : 1000×
- Perfect scaling needs all the program to be parallelized (*very* difficult)

Amdahl's law


1000
800
600
400
200
0
Speedup
Ideal
0
500
1000
1500
2000
Number of cores


99,9% parallel
0,1% sequential
1 core
x1.998
2 cores
x3.99
4 cores
x7.94
8 cores
x15.76
16 cores
x1000
∞ cores

Data access

Example: Coulomb Matrix

$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \iint \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

```
do mu=1,N
   do nu=1,N
      J(mu,nu) = 0.d0
      do la=1,N
         do si=1,N
            J(mu,nu) = J(mu,nu) + P(la,si) * ERI(mu,la,nu,si)
         end do
      end do
   end do
end do
```

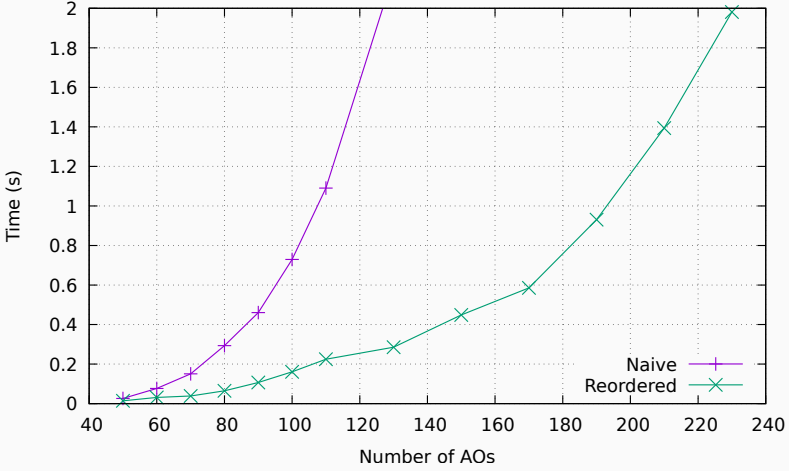
Example: Coulomb Matrix

$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \iint \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

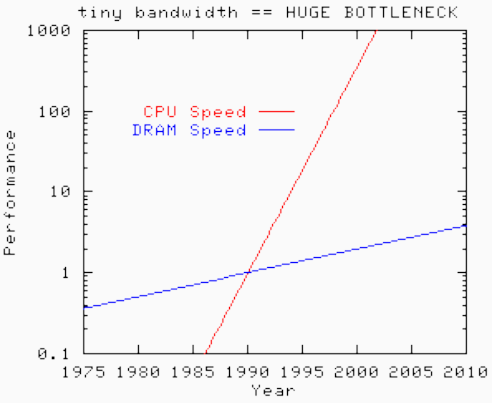
```
do nu=1,N       ! <-
   do mu=1,N    ! <-
      J(mu,nu) = 0.d0
      do si=1,N       ! <-
         do la=1,N    ! <-
            J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
         end do                                  !----------!
      end do
   end do
end do
```

We changed the order of indices in the storage of integrals, and the order of the loops.

Example: Coulomb Matrix



The memory wall



- CPU: $\times 1.55$ / year
- Memory: $\times 1.10$ / year

Stream benchmark :

`http://www.cs.virginia.edu/stream`

The memory wall

Bandwidth Throughput : Amount of data which goes through one point per unit of time

Latency Time to transfer *one* element between two points

	Latency	Bandwidth
	Low $(300km/h)^{-1}$	Low 1
	High $(80km/h)^{-1}$	High 68

The memory wall

Bandwidth Throughput : Amount of data which goes through one point per unit of time

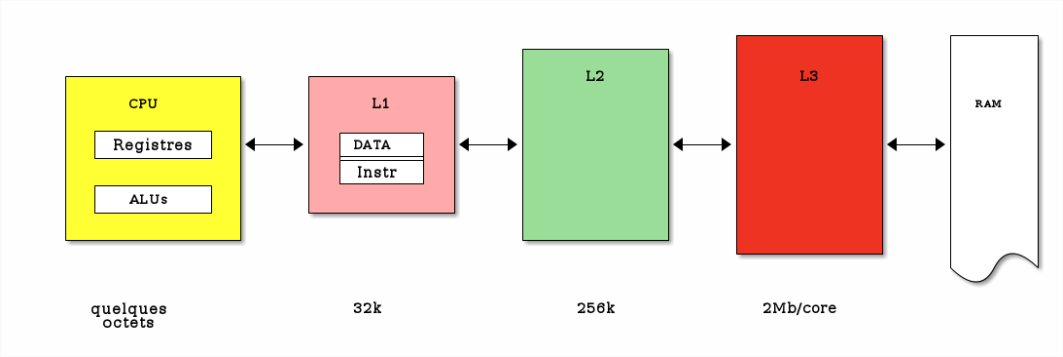
Latency Time to transfer *one* element between two points

Evolution over 20 years	Latency	Bandwidth
Disk	8×	143×
RAM	4×	120×
Ethernet	16×	1000×
CPU	21×	2250×

- Random access is more and more expensive (latency-bound)
- Hierarchical memories (caches) : hide latencies

Increasing the bandwidth

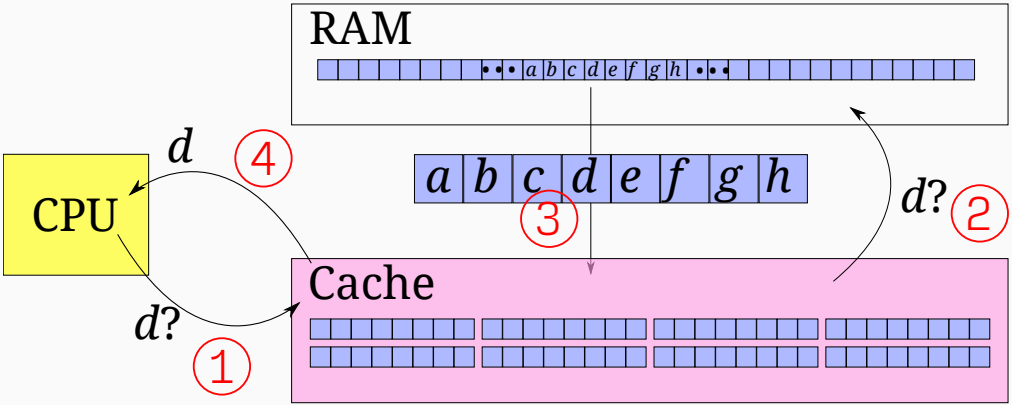
Hierarchical memories



- One ALU makes a `LOAD` d
- If $d \in$ L1, copy from d to the register
- If $d \notin$ L1,
- If $d \in$ L2, copy from d to L1 and in the register
- *etc*

Hierarchical memories

When a cache asks for some data at a higher level, it transfers a cache line : a block of fixed size (typically 64 bytes)



Hierarchical memories

Locality

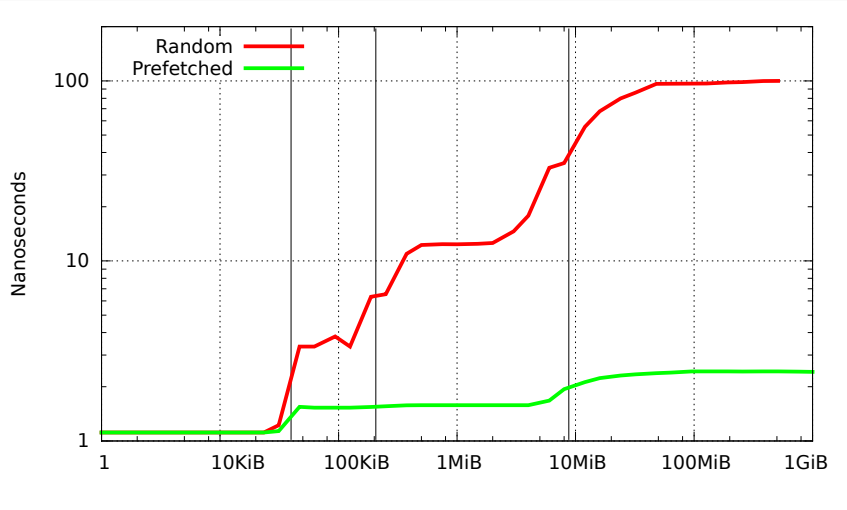
Spatial : If the CPU asks for e after having asked for d, e will be in the cache

Temporal : The cache line replaced by the new one is the *least recently used* (LRU)

Prefetching
If a regular access is detected, the next cache lines will be asked for in advance.

Latency (Nanoseconds)

Access in an increasingly large array:



Latency (Nanoseconds)

Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz

1 cycle = 0.28 ns, peak SP throughput (AVX2) = 0.0087 ns/flop

Integer	ADD	MUL	DIV	MOD	Bit
32 bit	0.28	0.84	6.60	7.07	0.28
64 bit	0.28	0.84	11.80	11.75	0.28
Floating Point	ADD	MUL	DIV		
32 bit	0.84	1.39	3.77		
64 bit	0.84	1.39	5.71		
Data read	Random	Prefetched			
L1	1.11	1.11			
L2	3.3	1.54			
L3	12.3	1.58			
RAM	100.	2.4			

`http://lmbench.sourceforge.net`

Latency (Cycles)

Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz

1 cycle = 0.28 ns, peak SP throughput (AVX2) = 32 flops/cycle

Integer	ADD	MUL	DIV	MOD	Bit
32 bit	1	3	23	25	1
64 bit	1	3	42	42	1
Floating Point	**ADD**	**MUL**	**DIV**		
32 bit	3	5	13		
64 bit	3	5	20		
Data read	**Random**	**Prefetched**			
L1	4	4			
L2	12	5.5			
L3	44	5.6			
RAM	357	8.6			

`http://lmbench.sourceforge.net`

Latency: Numbers to keep in mind

Operation	Latency (ns)	Scaled
Int ADD	0.3	1 s
FP ADD	0.9	3 s
FP MUL	1.5	5 s
Int32 DIV	6.6	22 s
FP64 DIV	5.7	19 s
L1 cache	1.2	4 s
L2 cache	3.5	12 s
L3 cache	13	43 s
RAM	79	4 min 23 s
Send 4KB with 100 Gbps Infiniband	1 040	57 min 46 s
Send 4KB over 10 Gbps ethernet	10 000	9 h 16 min
Write 4KB randomly to NVMe SSD	30 000	1 day 4 h
Transfer 1MB to/from PCIe GPU	80 000	3 days 11 h
Random Disk Access (seek+rotation)	10 000 000	1 year 21 days

Data access: summary

Accessing contiguous data: good!

- Maximizes bandwidth : you get a full cache line per transfer
- Minimizes latency : data is in cache + prefetch

Random access: bad!

- Minimizes bandwidth : you get one useful element per transfer
- Maximizes latency : cache miss + no prefetch

Data storage : 1D array

Fortran:
```
allocate (A(n))
```

C:
```
double* A = malloc (n*sizeof(double));
```

- `malloc` : Allocates a continuous block of memory
- `sizeof(double)` : number of bytes (8 for a `double`)

The elements of an array are contiguous

Dynamic allocation of 2D arrays

A is an $m \times n$ matrix

Solution 1 (bad)

```
double** a = malloc( m * sizeof(double*) ); // Allocate an array of m pointers

for (size_t i=0 ; i<n ; i++) { // Allocate m rows of size n
  a[i] = malloc( n * sizeof(double) );
}
```

- Two layers of indirection before accessing the data
- The rows don't necessarily follow each other in memory
  - Jumps in memory when going from one row to the next
  - Random jumps in memory when going through a column
- To free the matrix, we also need a loop

Dynamic allocation of 2D arrays

Solution 2

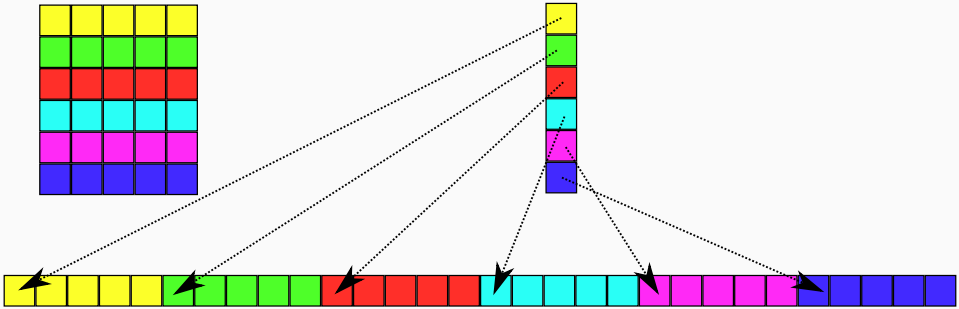
```
double* a = malloc( m*n * sizeof(double) );
#define a_(k,l) ( a[ (k)*n + (l) ] )

for (size_t i=0 ; i<m ; i++) {
  for (size_t j=0 ; j<n ; j++) {
    a_(i,j) = ...
  }
}
#undef a_
```

- Better performance
- Important for use with libraries (BLAS, MPI, ...)

Dynamic allocation of 2D arrays

Solution 3



Dynamic allocation of 2D arrays

Solution 3

```
double** a = malloc( m * sizeof(double*) ); // Allocate an array of m pointers
a[0] = malloc( m*n * sizeof(double) ); // Allocate a single block of memory

for (size_t i=1 ; i<m ; i++) {  // Populate pointers array
  a[i] = a[i-1] + n
}

for (size_t i=0 ; i<m ; i++) {
  for (size_t j=0 ; j<n ; j++) {
    a[i][j] = ...    // Use a[i][j] as usual
  }
}
```

Row-major and Column-major ordering

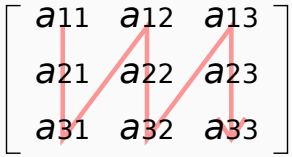
Fortran : Column-major

```
do j=1,n
  do i=1,n ! <-- i : inside j
     A(i,j)
  end do
end do
```

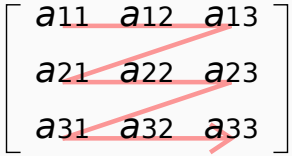
C : Row-major

```
for (i=0 ; i<n ; i++) {
  for (j=0 ; j<n ; j++) { // <-- j inside i
     A[i][j]
  }
}
```

Column-major



Row-major



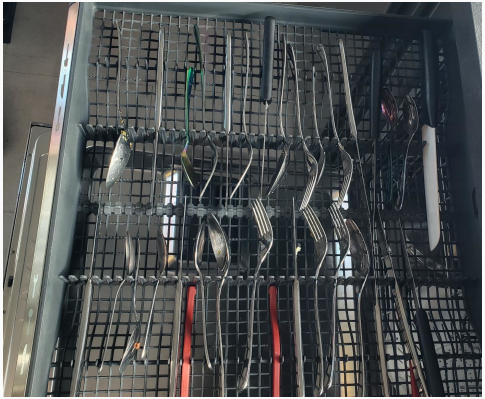
Re-ordering array indices

$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \int\int \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

```
do nu=1,N
 do mu=1,N
  J(mu,nu) = 0.d0
  do si=1,N
   do la=1,N
    J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
   end do
  end do
 end do
end do
```

Summary

Slow



Fast

Optimizing instructions

Beyond data layout

When should we optimize instructions?
When we are sure that the code is not memory-bound: computation is much faster than data movement (memory wall).

Arithmetic intensity: $\sigma = N(Flops)/N(bytes)$

- Add two vectors : `x[i] += y[i]`
  $\sigma = N/(24N) = 0.042$ flops/byte memory-bound
- Dot product : $x = \sum_i^N a_i \times b_i$
  $\sigma = 2N/(16N) = 0.125$ flops/byte memory-bound
- Matrix-vector: $X_i = \sum_j^N A_{ij} \times b_j$
  $\sigma = 2N^2/(8N^2 + 16N) \sim 0.25$ flops/byte memory-bound
- Matrix-matrix :$X_{ij} = \sum_k A_{ik} \times B_{kj}$
  $\sigma = 2N^3/(24N^2) \propto N$ CPU-bound

Instructions

Cheap instructions : high throughput, low latency

- ADD, MUL (int)
- ADD, MUL (float/double)
- AND, OR, XOR, ... (bool)

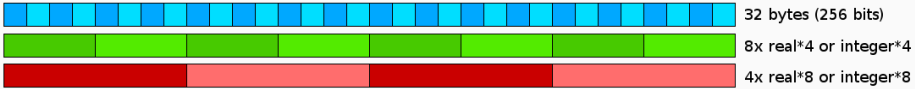
Expensive instructions : low throughput, high latency

- DIV, MOD (int)
- DIV (double)
- SQRT (double)
- EXP, POW, LOG, SIN, COS, etc (float/double)

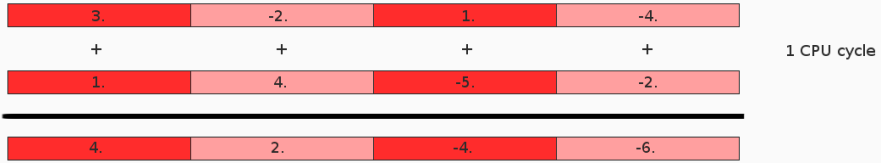
SIMD vectorization

SIMD: Single Instruction, Multiple Data

Execute the *same* instruction in parallel on all the elements of a vector:



Example : AVX vector ADD in double precision:



SIMD vectorization

Different SIMD instruction sets exist in the x86 micro-architecture:

- MMX : Integer (64-bit wide)
- SSE $\rightarrow$ SSE4.2 : Integer and Floating-point (128-bit)
- AVX, AVX2 : Integer and Floating-point (256-bit)
- AVX-512 : Integer and Floating-point (512-bit)

Requirements

- The elements of each SIMD vector must be contiguous in memory
- The first element of each SIMD vector must be *aligned* on a proper boundary (64, 128, 256 or 512-bit).

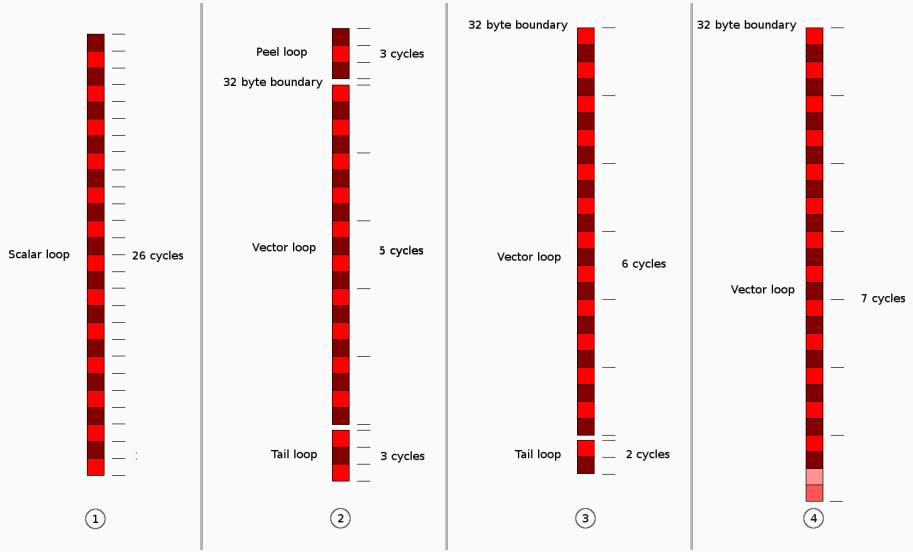
Automatic vectorization

The compiler can generate automatically vector instructions when possible. A double precision AVX auto-vectorized loop generates 3 loops:

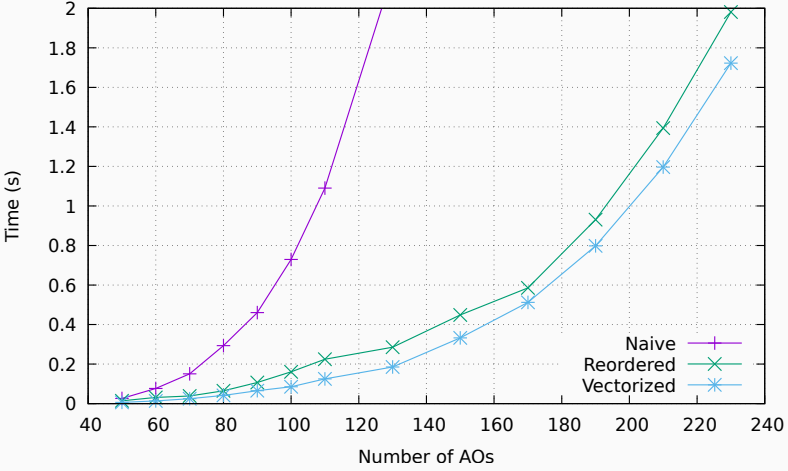
1. Peel loop (scalar): First elements until the 256-bit boundary is met
2. Vector loop: Vectorized version until the last vector of 4 elements
3. Tail loop (scalar): Last elements

Most efficient for large loop counts.

Automatic vectorization



Example: Coulomb Matrix



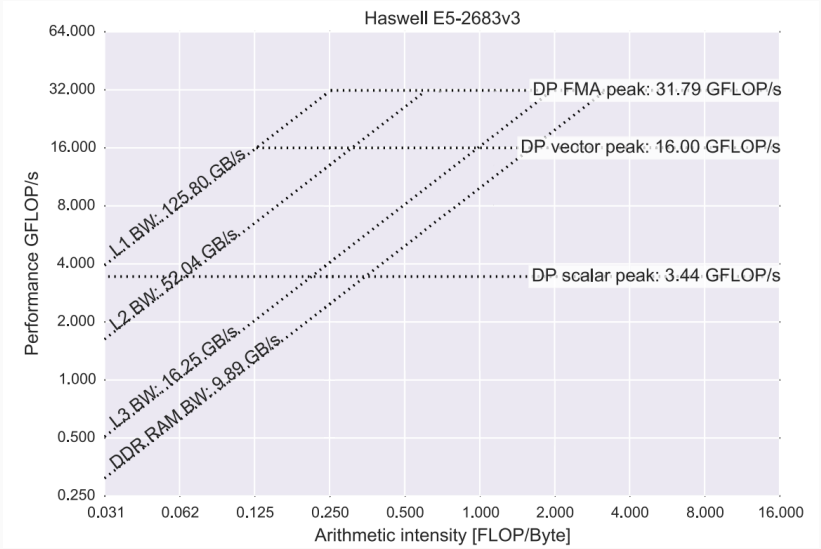
Example: Coulomb Matrix

$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \iint \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

```
do nu=1,N
   do mu=1,N
      J(mu,nu) = 0.d0
      do si=1,N
         do la=1,N
            J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
         end do
      end do
   end do
end do
```

- 2 N^4 flops, and N^2 stores (J) + N^2 (N^2 loads (P)) + N^4 loads (ERI)
- $\sigma = 2N^4/(8(N^2 + 2N^4)) \sim 0.125$

Roofline model



Loop unrolling

```
do nu=1,N
 do mu=1,N-4,4
  J(mu:mu+3,nu) = 0.d0
  do si=1,N
   do la=1,N
    J(mu+0,nu) = J(mu+0,nu) + P(la,si) * ERI(la,si,mu+0,nu)   ! P(la,si) is
    J(mu+1,nu) = J(mu+1,nu) + P(la,si) * ERI(la,si,mu+1,nu)   ! re-used 4x &
    J(mu+2,nu) = J(mu+2,nu) + P(la,si) * ERI(la,si,mu+2,nu)   ! 1 SIMD store
    J(mu+3,nu) = J(mu+3,nu) + P(la,si) * ERI(la,si,mu+3,nu)   ! for J
   end do
  end do
 end do
 do mu=N-4+1,N
  J(mu,nu) = 0.d0
  do si=1,N
   do la=1,N
    J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
   end do
  end do
 end do
end do
```

Loop unrolling

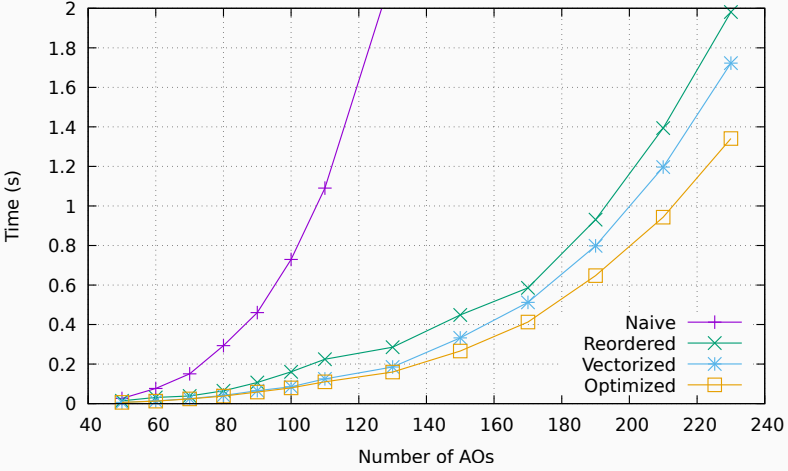
```
J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
```

- 2 loads
- 2 flops
- 2/16 = 0.125 flops/byte

```
J(mu+0,nu) = J(mu+0,nu) + P(la,si) * ERI(la,si,mu+0,nu) ! P(la,si) is
J(mu+1,nu) = J(mu+1,nu) + P(la,si) * ERI(la,si,mu+1,nu) ! re-used 4x &
J(mu+2,nu) = J(mu+2,nu) + P(la,si) * ERI(la,si,mu+2,nu) ! 1 SIMD store
J(mu+3,nu) = J(mu+3,nu) + P(la,si) * ERI(la,si,mu+3,nu) ! for J
```

- 1 load `P(la,si)` + 4 loads `ERI`
- 8 flops
- 8/40 = 0.2 flops/byte

Loop unrolling



Matrix multiplications

Why matrix multiplications?

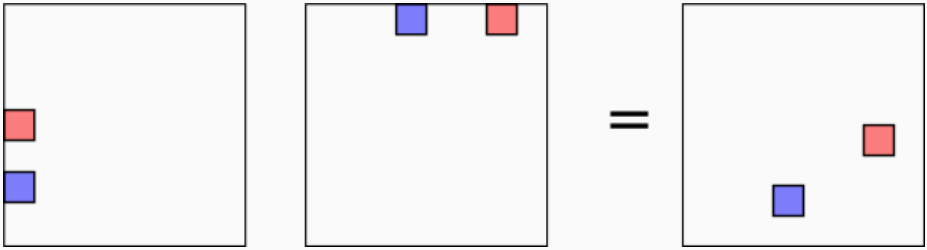
$$C_{mn} \leftarrow C_{mn} + A_{mk} \times B_{kn}$$

- Fused Multiply-Add (FMA) : $a \leftarrow a + b \times c$ is one SIMD instruction executed in 1 cycle
- A CPU has
  - Two independent SIMD FP units
  - Two independent SIMD memory load units
  - One memory store unit
- MM can be easily parallelized by performing the product in blocks

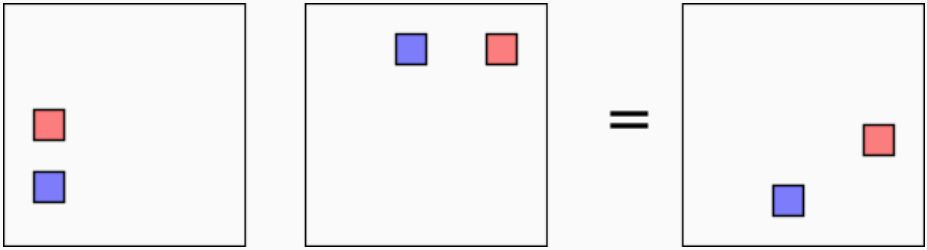
Arithmetic intensity

- Both CPUs and GPUs are well adapted compute matrix multiplications.
- DGEMM with vendor libraries can reach:
  - 90% of the peak on CPU, 80% of the peak on GPU

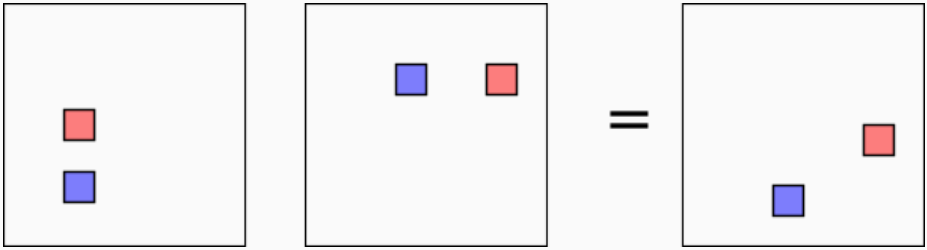
Blocking



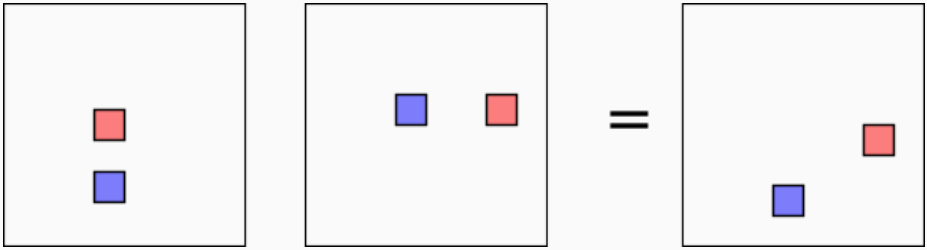
Blocking



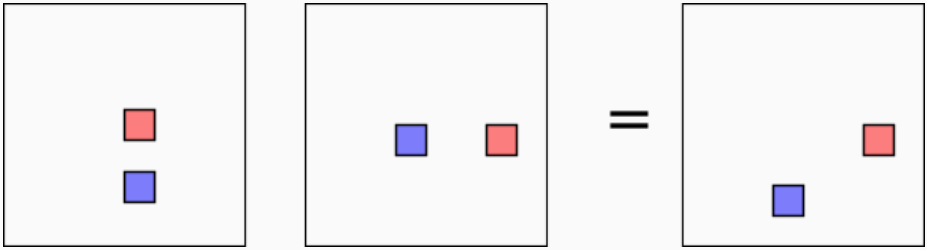
Blocking



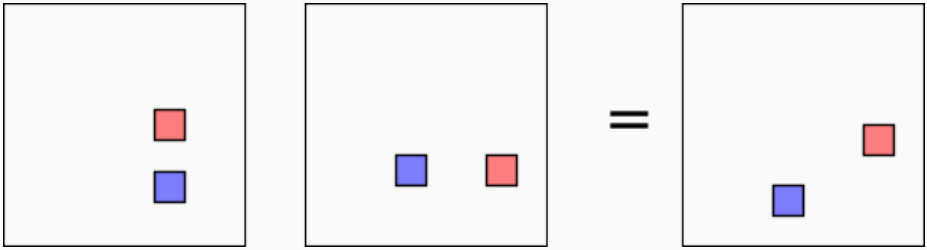
Blocking



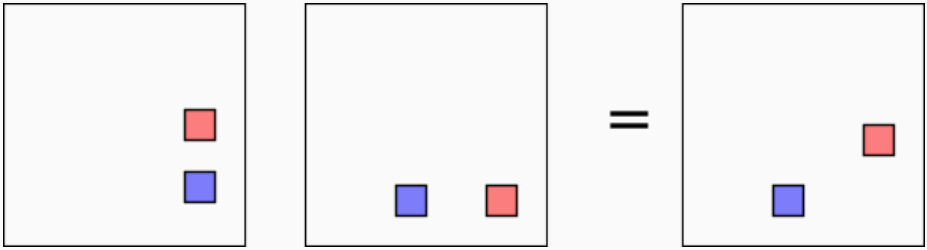
Blocking



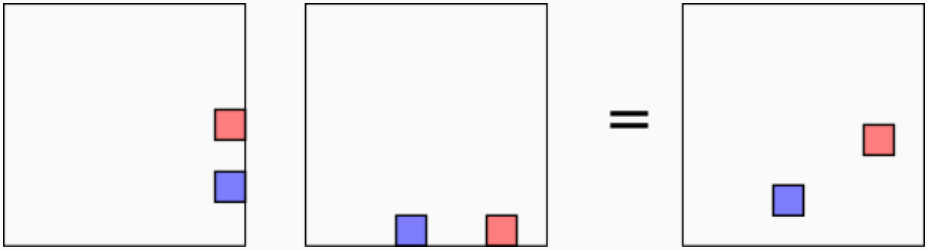
Blocking



Blocking



Blocking



Memory Layout

A vector / matrix and an array should not be confused:

- Vector / Matrix: Mathematical objects, with *logical* dimensions
- Array: a collection of same type data items that can be selected by indices. Dimensions are *physical*: they correspond to addresses in memory.

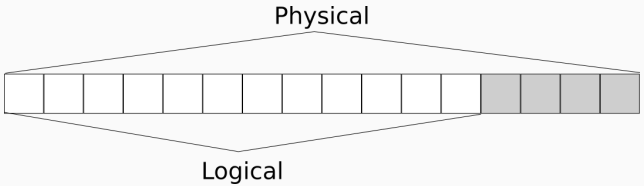
A matrix is stored in an array of FP numbers. The array has to be big enough to accept the matrix in it.

Example: Vector

- Vector : a_i, $1 \leq i \leq n$
- Array : Allocated memory: `A(1:nmax)`

```
double* A;
A = malloc (sizeof(double) * nmax);
```

```
double precision :: A(nmax)
do i=1,n
   A(i) = ...
end do
```



a_i = `A(i)` = `A[i-1]`

Example: Matrix

- Matrix : A_{ij}, $1 \leq i \leq m$, $1 \leq j \leq n$
- Array : Allocated memory: `A(1:mmax,1:nmax)`, `&(A[0])`

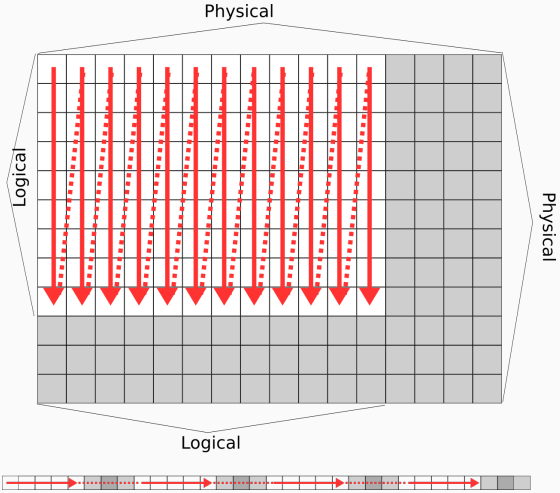
```
double* A;
A = malloc (sizeof(double) * nmax * mmax);
```

```
double** A;
A = malloc (sizeof(double*) * mmax);
A[0] = malloc (sizeof(double) * nmax * mmax);
for (size_t i=1 ; i< nmax ; i++)
  A[i] = A[i-1]+nmax;
```

```
double precision :: A(mmax,nmax)
do j=1,n
  do i=1,m
    A(i,j) = ...
  end do
end do
```

A_{ij} = `A(i,j) = A[ (j-1)*mmax + i-1 ]` or `A[j][i]`

Example: Matrix



Decrypting BLAS DGEMM arguments

$$C'_{mn} = \beta C_{mn} + \alpha \sum_k A_{mk} B_{kn}$$

- DGEMM : Double precision GEneral Matrix Matrix multiplication
- LDA : Leading dimension of A

Why pass LDA ? To compute the address in the array.

```
double precision :: A(LDA,*), B(LDB,*)
double precision :: C(LDC,*)

call DGEMM( 'N', 'N'    &  ! Transposed?
     , m, n, k          &  ! Dimensions
     , 1.d0             &  ! alpha
     , A, size(A,1)     &  ! A
     , B, size(B,1)     &  ! B
     , 0.d0             &  ! beta
     , C, size(C,1) )      ! C
```

Finding matrix multiplications by reshaping

- A rank-2 array can be reshaped into a rank-1 array
- A matrix can be interpreted as a vector
- A rank-3 array can be reshaped into a rank-2 or a rank-1 array

```
do j=1,n
   do i=1,n
      C(i,j) = 0.d0
      do l=1,n
         do k=1,n
            C(i,j) = C(i,j) + A(k,l,i) * B(k,l,j)
         end do
      end do
   end do
end do
```

- `A(k,l,i)` can be reshaped as: `A(kl,i)`
- `B(k,l,j)` can be reshaped as: `B(kl,j)`

Finding matrix multiplications by reshaping

This can be computed with a matrix multiplication:

$$C_{ij} = \sum_{kl} A^{\dagger}_{kl,i} B_{kl,j}$$

```
call DGEMM('T', 'N', n, n, (n*n), 1.d0, &
           A, size(A,1)*size(A,2), &
           B, size(B,1)*size(B,2), 0.d0, &
           C, size(C,1) )
```

Example: 4-index transformation

$$
\begin{aligned}
& \int \phi_\alpha(r_1)\phi_\beta(r_2)\frac{1}{|r_1 - r_2|}\phi_\gamma(r_1)\phi_\delta(r_2)\mathrm{d}r_1\mathrm{d}r_2 \\
= & \sum_{ijkl} B_{i\alpha}B_{j\beta}B_{k\gamma}B_{l\delta}\int \chi_i(r_1)\chi_j(r_2)\frac{1}{|r_1 - r_2|}\chi_k(r_1)\chi_l(r_2)\mathrm{d}r_1\mathrm{d}r_2 \\
C_{\alpha\beta\gamma\delta} = & \sum_{ijkl} A_{ijkl}\cdot B_{i\alpha}\cdot B_{j\beta}\cdot B_{k\gamma}\cdot B_{l\delta}
\end{aligned}
$$

- A : N^4 integrals in AO basis
- C : M^4 integrals in MO basis
- Conventional algorithm : Transform indices one by one : M^5 scaling

Example: 4-index transformation (Loops)

```
C(:,:,:,:)=0.d0
do s=1,N
   do l=1,N
      do k=1,N
         do j=1,N
            do i=1,N
               C(i,j,k,s) = C(i,j,k,s) + A(i,j,k,l) * B(l,s)
            end do
         end do
      end do
   end do
end do
```

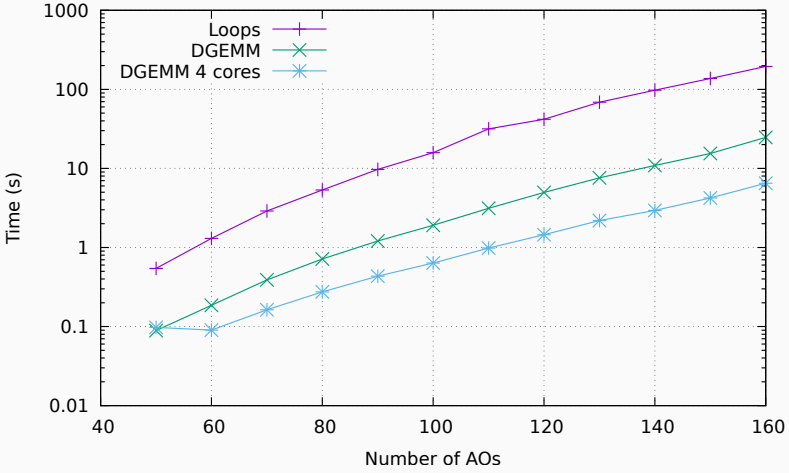
- Do one of such loop for each index

Example: 4-index transformation (DGEMM)

$$
\begin{aligned}
P_{jkl,\alpha} &\leftarrow \textstyle\sum_i A_{i,jkl} B_{i,\alpha} & \mathrm{P} &= \mathrm{A}^{\dagger}.\mathrm{B} \\
Q_{kl\alpha,\beta} &\leftarrow \textstyle\sum_j P_{j,kl\alpha} B_{j,\beta} & \mathrm{Q} &= \mathrm{P}^{\dagger}.\mathrm{B} \\
R_{l\alpha\beta,\gamma} &\leftarrow \textstyle\sum_k Q_{k,l\alpha\beta} B_{k,\gamma} & \mathrm{R} &= \mathrm{Q}^{\dagger}.\mathrm{B} \\
C_{\alpha\beta\gamma,\delta} &\leftarrow \textstyle\sum_l R_{l,\alpha\beta\gamma} B_{l,\delta} & \mathrm{C} &= \mathrm{R}^{\dagger}.\mathrm{B}
\end{aligned}
$$

```
call DGEMM('T','N', (nao*nao*nao), nmo, nao, 1.d0, A, nao, B, nao, 0.d0, P, (nao*nao*nao))
call DGEMM('T','N', (nao*nao*nmo), nmo, nao, 1.d0, P, nao, B, nao, 0.d0, Q, (nao*nao*nmo))
call DGEMM('T','N', (nao*nmo*nmo), nmo, nao, 1.d0, Q, nao, B, nao, 0.d0, R, (nao*nmo*nmo))
call DGEMM('T','N', (nmo*nmo*nmo), nmo, nao, 1.d0, R, nao, B, nao, 0.d0, C, (nmo*nmo*nmo))
```

Example: 4-index transformation



Other examples

APPENDIX: DENSITY MATRICES AND ORBITAL HESSIAN

For completeness, we include here expressions for the pCCD density matrices and orbital rotation Hessian; together with the orbital rotation gradient of Eq. (25), these provide everything needed for the Newton-Raphson algorithm we use for orbital optimization.

Recall that the energy is written as

$$\mathcal{E}(\kappa) = \langle 0|(1+Z)\, \mathrm{e}^{-T}\, \mathrm{e}^{-\kappa}\, H\, \mathrm{e}^{\kappa}\, \mathrm{e}^{T}|0\rangle \tag{A1}$$

with

$$\kappa = \sum_{p>q} \sum_{\sigma} \kappa_{pq} \left(c^{\dagger}_{p_\sigma} c_{q_\sigma} - c^{\dagger}_{q_\sigma} c_{p_\sigma} \right), \tag{A2}$$

where the orbital rotation is given by the unitary transformation $\exp(\kappa)$. At every step of the Newton-Raphson scheme,

$$\left.\frac{\partial \mathcal{E}(\kappa)}{\partial \kappa_{pq}}\right|_{\kappa=0} = \mathcal{P}_{pq} \sum_{\sigma} \langle [H, c^{\dagger}_{p_\sigma} c_{q_\sigma}]\rangle, \tag{A3}$$

where $\mathcal{P}_{pq}$ is a permutation operator $\mathcal{P}_{pq} = 1 - (p \leftrightarrow q)$ and the notation for the expectation value means

$$\langle O\rangle = \langle 0|(1+Z)\, \mathrm{e}^{-T}\, O\, \mathrm{e}^{T}|0\rangle. \tag{A4}$$

Similarly, the Hessian is

$$\begin{aligned} H_{pq,rs} &= \left.\frac{\partial^2 \mathcal{E}(\kappa)}{\partial \kappa_{pq}\, \partial \kappa_{rs}}\right|_{\kappa=0} \\ &= \frac{1}{2}\, \mathcal{P}_{pq}\, \mathcal{P}_{rs} \sum_{\sigma,\eta} \langle [[H, c^{\dagger}_{p_\sigma} c_{q_\sigma}], c^{\dagger}_{r_\eta} c_{s_\eta}]\rangle \\ &\quad + \frac{1}{2}\, \mathcal{P}_{pq}\, \mathcal{P}_{rs} \sum_{\sigma,\eta} \langle [[H, c^{\dagger}_{r_\eta} c_{s_\eta}], c^{\dagger}_{p_\sigma} c_{q_\sigma}]\rangle, \end{aligned} \tag{A5}$$

where η is another spin index. We obtain

$$\begin{aligned} H_{pq,rs} = \mathcal{P}_{pq}\, \mathcal{P}_{rs} \Bigg\{ & \frac{1}{2} \sum_{u} \left[\delta_{qr} \left(h^{u}_{p}\, \gamma^{s}_{u} + h^{s}_{u}\, \gamma^{u}_{p} \right) + \delta_{ps} \left(h^{u}_{r}\, \gamma^{q}_{u} + h^{q}_{u}\, \gamma^{u}_{r} \right) \right] - \left(h^{s}_{p}\, \gamma^{q}_{r} + h^{q}_{r}\, \gamma^{s}_{p} \right) \\ & + \frac{1}{2} \sum_{tuv} \left[\delta_{qr} \left(v^{uv}_{pt}\, \Gamma^{st}_{uv} + v^{st}_{uv}\, \Gamma^{uv}_{pt} \right) + \delta_{ps} \left(v^{qt}_{uv}\, \Gamma^{uv}_{rt} + v^{uv}_{rt}\, \Gamma^{qt}_{uv} \right) \right] \\ & + \sum_{uv} \left(v^{uv}_{pr}\, \Gamma^{qs}_{uv} + v^{qs}_{uv}\, \Gamma^{uv}_{pr} \right) - \sum_{tu} \left(v^{st}_{pu}\, \Gamma^{qu}_{rt} + v^{ts}_{pu}\, \Gamma^{qu}_{tr} + v^{qu}_{rt}\, \Gamma^{st}_{pu} + v^{qu}_{tr}\, \Gamma^{ts}_{pu} \right) \Bigg\}. \end{aligned} \tag{A6}$$

Henderson *et al*, http://dx.doi.org/10.1063/1.4904384

Summary

- Organize data for stride-1 access
- Favor good access to writing
- Consider matrix multiplication as *the magic instruction for performance*
- Porting code based on DGEMM to GPU can be done with minimal effort: use MAGMA, cuBLAS or Chameleon instead of BLAS

Exercise

Matrix multiplication

- Write in C functions that compute, in double precision:
  - $C = A.B^{\dagger}$
  - $C = A^{\dagger}.B$
- Time your two functions (see next slide) with matrices $1\,000 \times 1\,000$, compiled with `-O0`, `-Ofast` and `-Ofast -march=native`.
- Do you see a difference? What can explain it?
- Now call DGEMM from the MKL implementation of BLAS. Is it faster?

```
icx -qmkl=sequential my_code.c -o my_code
```

How to time a function

```c
#include <time.h>

clock_t start, end;
double cpu_time_used;

start = clock(); // Record the starting time
f();             // Call the function you want to time
end = clock();   // Record the ending time

// Calculate the elapsed time in seconds
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
```

Solution i

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cblas.h>
#define N 1000

// A.B^t
void abt(double** A, double** B, double** C, size_t n) {
    for (size_t i=0 ; i<n ; i++) {
       for (size_t j=0 ; j<n ; j++) {
           C[i][j] = 0.;
           for (size_t k=0 ; k<n ; k++) {
               C[i][j] += A[i][k] * B[j][k];
           }
       }
```

Solution ii

```
    }
}

// A^t.B
void atb(double** A, double** B, double** C, size_t n) {
    for (size_t i=0 ; i<n ; i++) {
        for (size_t j=0 ; j<n ; j++) {
            C[i][j] = 0.;
            for (size_t k=0 ; k<n ; k++) {
                C[i][j] += A[k][i] * B[k][j];
            }
        }
    }
}
```

Solution iii

```
int main() {
    clock_t start, end;
    double cpu_time_used;

    // Allocate matrices

    double** A  = malloc(N*sizeof(double*));
    double** B  = malloc(N*sizeof(double*));
    double** C1 = malloc(N*sizeof(double*));
    double** C2 = malloc(N*sizeof(double*));

    A [0] = malloc(N*N*sizeof(double));
    B [0] = malloc(N*N*sizeof(double));
    C1[0] = malloc(N*N*sizeof(double));
    C2[0] = malloc(N*N*sizeof(double));
```

Solution iv

```
for (size_t i=1 ; i<N ; i++) {
  A[i] = A[i-1]+N;
  B[i] = B[i-1]+N;
  C1[i] = C1[i-1]+N;
  C2[i] = C2[i-1]+N;
}


// Fill matrices with some numbers

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

Solution v

```
// Measure A^t.B
start = clock();
atb(A, B, C1, N);
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("A^t.B       : %f seconds\n", cpu_time_used);

// Measure A^t.B with BLAS
start = clock();
cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, N,N,N,
            1.0, A[0], N, B[0], N, 0.0, C2[0], N);
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("A^t.B (BLAS): %f seconds\n", cpu_time_used);
```

Solution vi

```
// Check that C1 and C2 match: Compute ||C1-C2||^2
double norm2 = 0.;
for (size_t i=0 ; i<N ; i++) {
  for (size_t j=0 ; j<N ; j++) {
    double delta = C1[i][j] - C2[i][j];
    norm2 += delta*delta;
  }
}
printf("||C1-C2||^2 = %e\n", norm2);


// Measure A.B^t
start = clock();
abt(A, B, C1, N);
end = clock();
```

Solution vii

```
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("A.B^t       : %f seconds\n", cpu_time_used);

// Measure A.B^t with BLAS
start = clock();
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, N,N,N,
            1.0, A[0], N, B[0], N, 0.0, C2[0], N);
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("A^t.B (BLAS): %f seconds\n", cpu_time_used);

// Check that C1 and C2 match: Compute ||C1-C2||^2
norm2 = 0.;
for (size_t i=0 ; i<N ; i++) {
  for (size_t j=0 ; j<N ; j++) {
    double delta = C1[i][j] - C2[i][j];
```

Solution viii

```
        norm2 += delta*delta;
      }
    }
    printf("||C1-C2||^2 = %e\n", norm2);

    // Clean up

    free(A[0]);  free(A);
    free(B[0]);  free(B);
    free(C1[0]); free(C1);
    free(C2[0]); free(C2);

    return 0;
}
```