# High Performance Computing and aspects of Computing in Lattice Gauge Theories

\\

Giannis Koutsou
Computation-based Science and Technology Research Centre (CaSToRC)
The Cyprus Institute

\\

EuroPLEx School, 29th October 2020

## Lecture 1: Introduction to HPC

**29th October**

- Introduction to high performance computing (HPC) and parallel computing
    - Scalability and other performance metrics
    - Modern HPC architectures and their characteristics
- Supercomputing ecosystem
    - European landscape
    - Exa-scale initiatives
- Considerations when optimizing scientific codes
    - Simple performance models
    - Outline of select (on-node) optimization strategies

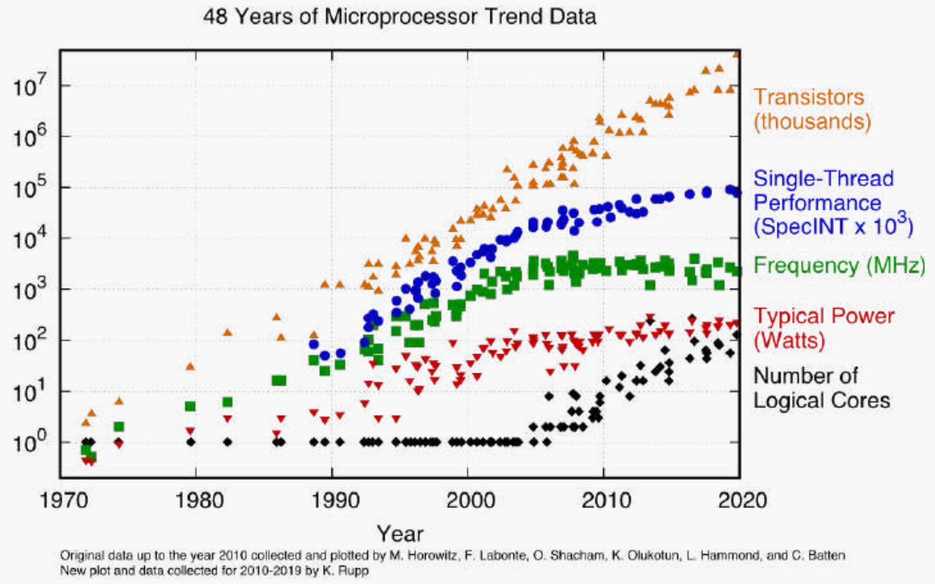
## High performance computing

**High performance computing means parallel computing**

# High performance computing

**High performance computing means parallel computing**

- Computers are becoming all the more parallel due to technological constraints

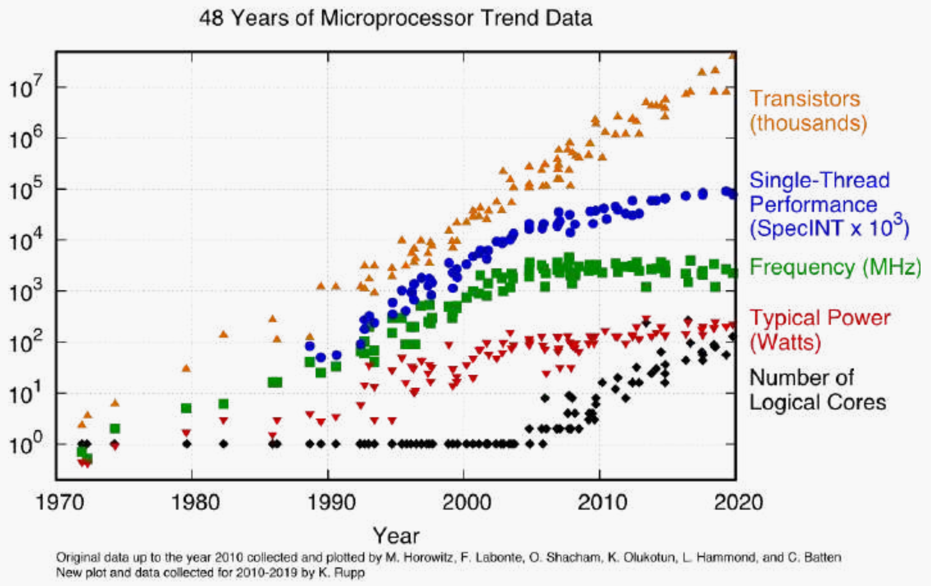


https://github.com/karlrupp/microprocessor-trend-data

# High performance computing

**High performance computing means parallel computing**

- Computers are becoming all the more parallel due to technological constraints



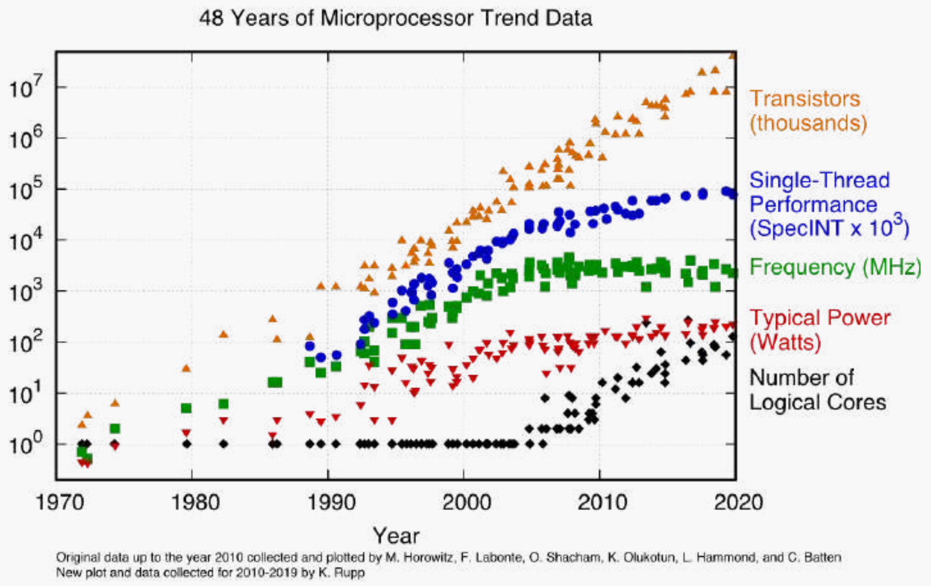
https://github.com/karlrupp/microprocessor-trend-data

- Moore's law: transistor count exponentially increasing (but not as originally expected)

# High performance computing

**High performance computing means parallel computing**

- Computers are becoming all the more parallel due to technological constraints



https://github.com/karlrupp/microprocessor-trend-data

- Moore's law: transistor count exponentially increasing (but not as originally expected)
- Dennard Scaling: $P \propto AfV^2$. Lost around 2006 ($P$: power, $A$: area, $f$: freq., $V$: voltage)

## High performance computing

**High performance computing means parallel computing**

- Exploiting parallelism is essential for scientific computing

## High performance computing

**High performance computing means parallel computing**

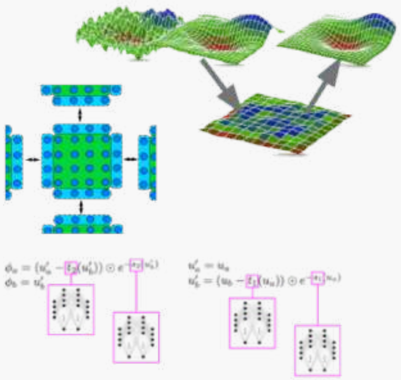
- Exploiting parallelism is essential for scientific computing
- Practitioners of computational sciences benefit from knowledge of concepts and challenges of parallel computing

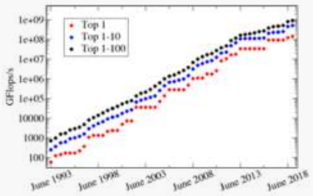
# High performance computing

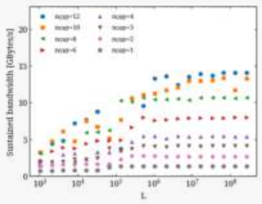
**High performance computing means parallel computing**

- Exploiting parallelism is essential for scientific computing
- Practitioners of computational sciences benefit from knowledge of concepts and challenges of parallel computing
    - Architectures and their characteristics
    - Algorithms and how amenable they are to parallelisation
    - Performance metrics and their significance, e.g. sustained and peak floating point performance, bandwidth, scalability

**Architectures** | **Algorithms** | **Performance metrics**







## High performance computing

**How do we "spend" parallelism?**

- *Capacity computing*
  - Improve time-to-solution of a problem that can also run on less number of processes
  - E.g. solve many small problems
- *Capability computing*
  - Solve a problem that was *impossible* to solve on less processes
  - E.g. solve a problem using $\mathrm{N}$ nodes, that cannot fit in memory of less nodes

## High performance computing

**How do we "spend" parallelism?**

- *Capacity computing*
    - Improve time-to-solution of a problem that can also run on less number of processes
    - E.g. solve many small problems
- *Capability computing*
    - Solve a problem that was *impossible* to solve on less processes
    - E.g. solve a problem using $\mathrm{N}$ nodes, that cannot fit in memory of less nodes

---

*High Throughput Computing* sometimes used to identify capacity computing, with HPC used to mean capability computing

## High performance computing

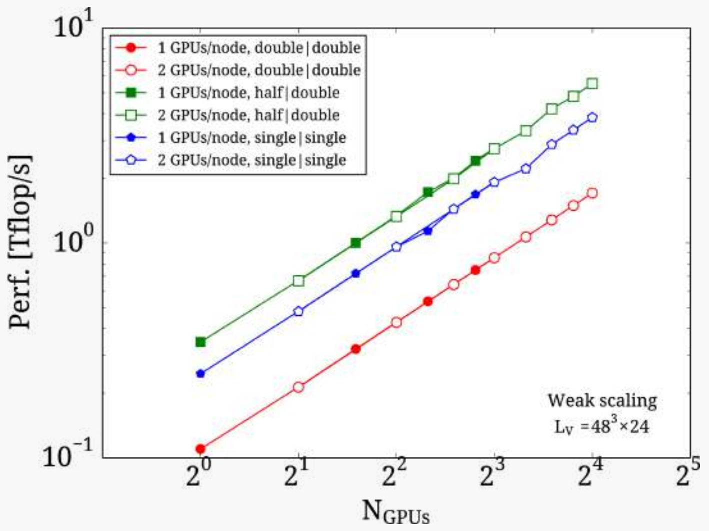
**Concepts of parallel computing**

- **Scalability**: The rate at which time-to-solution improves as we increase processing units
- **Weak scaling**: Increase processing units; keep the **local** problem size fixed $\Rightarrow$ for increasing global problem size
- **Strong scaling**: Increase processing units; keep the **global** problem size fixed $\Rightarrow$ for decreasing local problem size

## High performance computing

**Concepts of parallel computing**

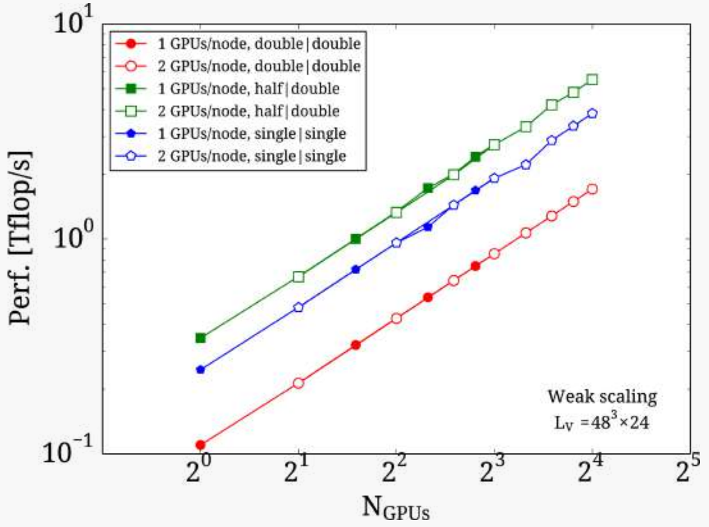
- **Scalability**: The rate at which time-to-solution improves as we increase processing units
- **Weak scaling**: Increase processing units; keep the **local** problem size fixed $\Rightarrow$ for increasing global problem size
- **Strong scaling**: Increase processing units; keep the **global** problem size fixed $\Rightarrow$ for decreasing local problem size

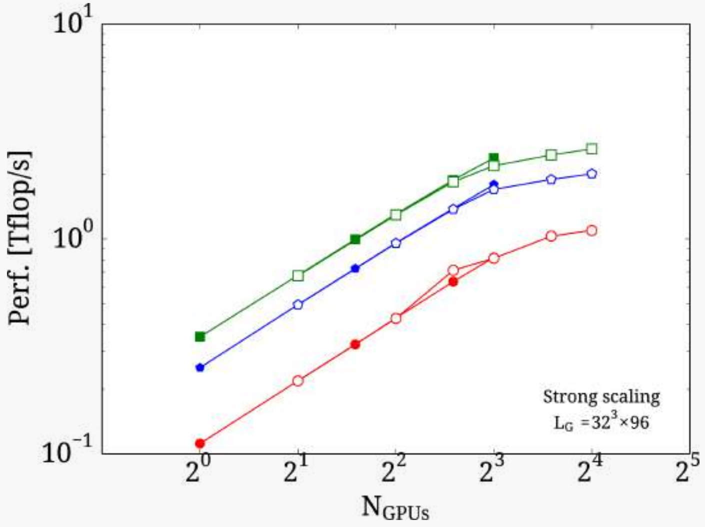


# High performance computing

**Concepts of parallel computing**

- **Scalability**: The rate at which time-to-solution improves as we increase processing units
- **Weak scaling**: Increase processing units; keep the **local** problem size fixed $\Rightarrow$ for increasing global problem size
- **Strong scaling**: Increase processing units; keep the **global** problem size fixed $\Rightarrow$ for decreasing local problem size





# High performance computing

**Concepts of parallel computing**

- **Scalability**: The rate at which time-to-solution improves as we increase processing units
- Quantify scalability: Speed-up ($S$) when using $N$ processes

$$S(N) = \frac{T_0}{T(N)}$$

  - $T_0$: Reference time-to-solution
  - $T(N)$: Time-to-solution when using $N$ processes

# High performance computing

**Concepts of parallel computing**

- **Scalability**: The rate at which time-to-solution improves as we increase processing units
- Quantify scalability: Speed-up ($S$) when using $N$ processes

$$S(N) = \frac{T_0}{T(N)}$$

  - $T_0$: Reference time-to-solution
  - $T(N)$: Time-to-solution when using $N$ processes

- Quantify divergence of scalability from ideal: *parallel efficiency*

$$\epsilon(N) = S(N)\frac{N_0}{N}$$

  - Ideal scaling: $\epsilon(N) \sim 1$
  - Typically (e.g. in proposals for computer time) an application is considered "scalable" in the region of $N$ for which $\epsilon(N) \geqslant 0.5$

# High performance computing

## Concepts of parallel computing

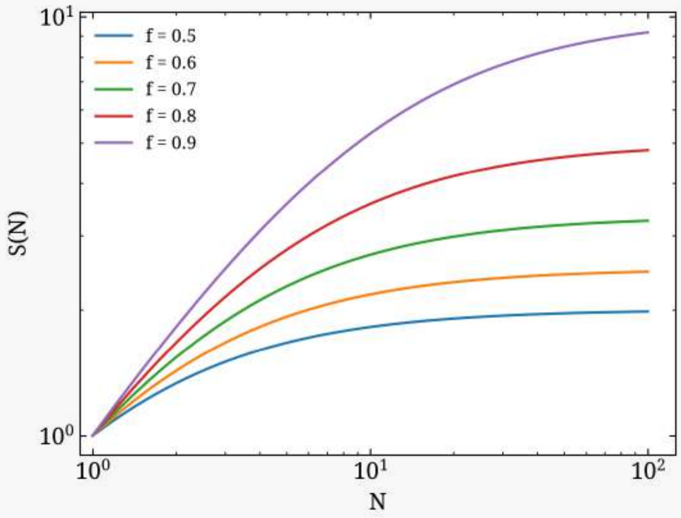
- **Amdahl's Law**: A simple model for the expected scalability of an arbitrary application
  - $f$: fraction of application that can be parallelized
  - $T_0$: time-to-solution of code when using one process
  - $N$: Number of processes

- Time-to-solution as a function of $N$:

$$T(N) = (1-f)T_0 + f\frac{T_0}{N}$$

- Speed-up as a function of $N$

$$S(N) = \frac{T_0}{T(N)} = \frac{1}{1-f+\frac{f}{N}}$$



## High performance computing

**Concepts of parallel computing**

- Performance metrics
    - Floating point rate: number of floating point operations carried out by a computational task per unit time
    - I/O or Bandwidth: bytes read and written per unit time

# High performance computing

**Concepts of parallel computing**

- Performance metrics
  - Floating point rate: number of floating point operations carried out by a computational task per unit time
  - I/O or Bandwidth: bytes read and written per unit time
- Distinguish between theoretical peak and sustained
  - Theoretical peak: Assuming full utilization of hardware
  - Sustained: measured, e.g. via running an application

## High performance computing

**Concepts of parallel computing**

- Performance metrics
    - Floating point rate: number of floating point operations carried out by a computational task per unit time
    - I/O or Bandwidth: bytes read and written per unit time
- Distinguish between theoretical peak and sustained
    - Theoretical peak: Assuming full utilization of hardware
    - Sustained: measured, e.g. via running an application
- Note that usually a single floating point operation (flop) is an `add`, `sub`, or `mul`, with other operations (e.g. `dev`, exponentiation, etc.) typically requiring more than 1 flop.

## High performance computing

**Concepts of parallel computing**

Taxonomy of computer architectures, *Flynn's Taxonomy*:

- Single Instruction stream, Single Data stream (SISD)
- Multiple Instruction streams, Single Data stream (MISD)
- Single Instruction stream, Multiple Data streams (SIMD)
- Multiple Instruction streams, Multiple Data streams (MIMD)

# High performance computing

**Concepts of parallel computing**

Taxonomy of computer architectures, *Flynn's Taxonomy*:

- Single Instruction stream, Single Data stream (SISD)
- Multiple Instruction streams, Single Data stream (MISD)
- Single Instruction stream, Multiple Data streams (SIMD)
- Multiple Instruction streams, Multiple Data streams (MIMD)

Parallel computing

- Most computing devices have underlying SIMD architecture units: GPUs, CPUs, etc.
- Most supercomputers can be considered MIMD architectures: multiple interconnected computing devices that can issue the same or different instructions on multiple data

## High performance computing

**What's in a supercomputer?**

- Compute Nodes
    - Computational units - CPU and potentially a co-processor, e.g. a GPU
    - Memory (i.e. RAM)
    - Some storage and/or NVMe
    - Network interfaces, possibly separate between management and workload

## High performance computing

**What's in a supercomputer?**

- Compute Nodes
  - Computational units - CPU and potentially a co-processor, e.g. a GPU
  - Memory (i.e. RAM)
  - Some storage and/or NVMe
  - Network interfaces, possibly separate between management and workload
- Interconnect
  - Interfaces on nodes
  - Wiring and switches

## High performance computing

**What's in a supercomputer?**

- Compute Nodes
  - Computational units - CPU and potentially a co-processor, e.g. a GPU
  - Memory (i.e. RAM)
  - Some storage and/or NVMe
  - Network interfaces, possibly separate between management and workload
- Interconnect
  - Interfaces on nodes
  - Wiring and switches
- Storage
  - Still predominantly spinning disks
  - Solid state drives are emerging for smaller scratch space
  - Tape systems for archiving

## High performance computing

**What's in a supercomputer?**

- Compute Nodes
  - Computational units - CPU and potentially a co-processor, e.g. a GPU
  - Memory (i.e. RAM)
  - Some storage and/or NVMe
  - Network interfaces, possibly separate between management and workload
- Interconnect
  - Interfaces on nodes
  - Wiring and switches
- Storage
  - Still predominantly spinning disks
  - Solid state drives are emerging for smaller scratch space
  - Tape systems for archiving
- Front-end nodes
  - For user access
  - Compiling, submitting jobs, etc.

## High performance computing

**Computing devices**

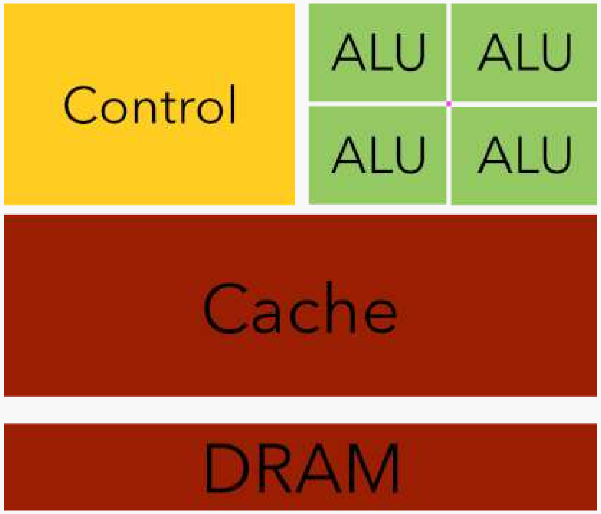
Most supercomputers have compute nodes equipped with a co-processor, typically a general purpose GPU

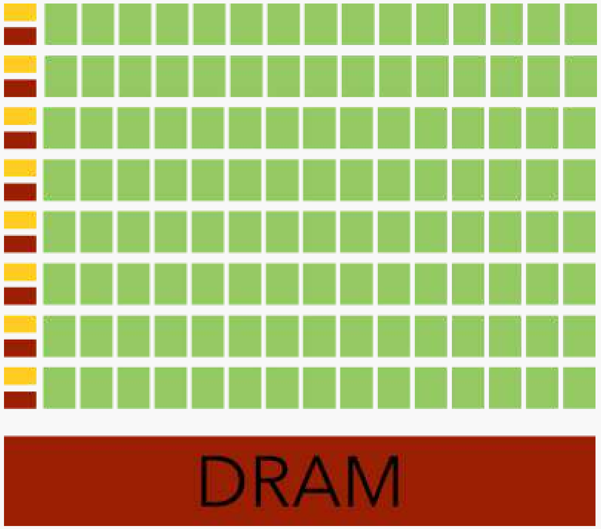
- CPU architecture
    - Optimized for handling a diverse ranged of instructions on multiple data
    - Large caches per core
    - Smaller bandwidth to memory but typically larger memory
    - Reliance on prefetching
    - Some availability of SIMD floating point units
- GPU architecture
    - Optimized for *throughput*, i.e. applying the same operation on multiple data
    - Smaller caches per "core"
    - Higher bandwidth to memory but typically smaller memory
    - Reliance on very wide SIMD units

## High performance computing

**Computing devices**

Most supercomputers have compute nodes equipped with a co-processor, typically a general purpose GPU





### CPU

- Large area devoted to control
- Large caches
- Few ALUs

### GPU

- Less area devoted to control
- Small caches
- Most area dedicated to ALUs

## High performance computing

**Computing devices**

CPU architectures and main characteristics

- Intel Xeon and AMD EPYC (x86), various ARM implementations, and IBM Power
- $O(10)$ cores per CPU (current max: 64), 2 - 4 CPUs per node
- Memory bandwidth of $\sim$50-100 GBytes/s
- Theoretical peak floating point performance $\sim$30-50 Gflop/s per core

GPU architectures and main characteristics

- NVIDIA Tesla and AMD Radeon
- $O(1000)$ "cores" or Arithmetic and Logical Unit (ALUs). 2 - 6 GPUs per node
- Memory bandwidth of $O(1000)$ GBytes/s
- Theoretical peak floating point performance $\sim$10-20 Tflop/s per GPU

## High performance computing

**Computing devices**

CPU architectures and main characteristics

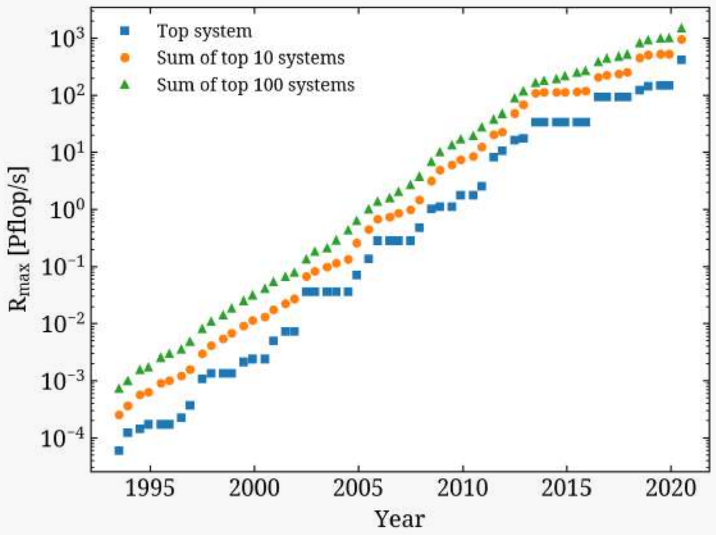
- Intel Xeon and AMD EPYC (x86), various ARM implementations, and IBM Power
- $\mathcal{O}(10)$ cores per CPU (current max: 64), 2 - 4 CPUs per node
- Memory bandwidth of $\sim$50-100 GBytes/s
- Theoretical peak floating point performance $\sim$30-50 Gflop/s per core

GPU architectures and main characteristics

- NVIDIA Tesla and AMD Radeon
- $\mathcal{O}(1000)$ "cores" or Arithmetic and Logical Unit (ALUs). 2 - 6 GPUs per node
- Memory bandwidth of $\mathcal{O}(1000)$ GBytes/s
- Theoretical peak floating point performance $\sim$10-20 Tflop/s per GPU

Intel Xe "Ponte Vecchio" GPUs announced for 2021

# Worldwide landscape of supercomputers



- Bi-annual ranking of supercomputers
- Classification according to *sustained* floating point performance
- Run High Performance Linpack (HPL) on whole system

- `https://www.top500.org` Latest list: **June 2020**
    - Top system: Japan, ARM-based system, 415 Pflop/s
    - Second: US, GPUs (NVIDIA V100), 200 Pflop/s
    - Top in Europe: IT, sixth overall, GPUs (NVIDIA V100), 35 Pflop/s
- 6 out of top 10 equipped with GPUs

## Supercomputing landscape

**Notable supercomputers in Europe**

**Marconi100** - CINECA, IT

- IBM Power CPUs, NVIDIA V100 GPUs
- $\sim$30 Pflop/s theoretical peak

**Piz Daint** - CSCS, CH

- Cray XC30, Intel Xeon CPUs, NVIDIA P100 GPUs
- $\sim$27 Pflop/s theoretical peak

## Supercomputing landscape

**Notable supercomputers in Europe**

**SuperMUC** - LRZ, DE

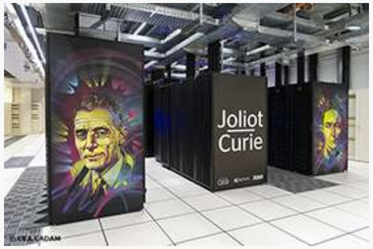
- IBM ThinkSystem, Intel Xeon CPUs
- ~27 Pflop/s theoretical peak

**Juwels** - JSC, DE

- Bull Sequana, Intel Xeon CPUs, NVIDIA A100 GPUs
- ~70 Pflop/s theoretical peak (with GPU partition, currently being deployed)

# Supercomputing landscape

**Notable supercomputers in Europe**



**Joliet/Curie** - CEA, FR

- Bull Sequana, AMD EPYC CPUs
- ∼12 Pflop/s theoretical peak

**HAWK** - HLRS, DE

- HPE Apollo, AMD EPYC CPUs
- ∼26 Pflop/s theoretical peak

**MareNostrum-4** - BSC, ES

- Mainly Lenovo, Intel Xeon Platinum plus smaller ARM, IBM Power9 w/ NVIDIA V100, and AMD partitions
- ∼14 Pflop/s theoretical peak

## Supercomputer access

**In Europe, these systems also made available via a single, centralized allocation process**

So-called Tier-0 PRACE access

- Technical review: need to show that methods and software are appropriate for the underlying architecture. Scaling and performance analysis required.
- Scientific review: peer-review of the proposed science.
- $O(10)$-$O(100)$ core-hours for individual projects

Smaller-scale access available nationally

- Depends on national mechanisms for access
- Usually follows same approach of technical and scientific review

## Supercomputer access

**In Europe, these systems also made available via a single, centralized allocation process**

So-called Tier-0 PRACE access

- Technical review: need to show that methods and software are appropriate for the underlying architecture. Scaling and performance analysis required.
- Scientific review: peer-review of the proposed science.
- $O(10)$-$O(100)$ core-hours for individual projects

Smaller-scale access available nationally

- Depends on national mechanisms for access
- Usually follows same approach of technical and scientific review

Access to such resources requires good understanding of HPC and the challenges in achieving efficient software implementations

## Supercomputing landscape

**European strategy**

Up until ~2020

- Countries procure systems using own funds
- EU supports operations and distribution of computational resources via projects (e.g. PRACE)
- EU supports prototyping for future systems (e.g. DEEP and Mont Blanc)

From 2020: EuroHPC Joint Undertaking

- EU co-funds procurement of supercomputers by consortia of European states
- Three pre-exascale systems will be deployed by 2021
- Two exascale systems by 2022 or 2023

## Supercomputing landscape

**EuroHPC Pre-exascale systems, coming on-line 2021**

LUMI

- Hosted at CSC (FI), partners BE, CZ, DK, EE, IS, NO, PL, SE, CH
- HPE, AMD EPYC CPUs and AMD Instinct GPUs
- Peak: 552 Pflop/s

## Supercomputing landscape

**EuroHPC Pre-exascale systems, coming on-line 2021**

LUMI

- Hosted at CSC (FI), partners BE, CZ, DK, EE, IS, NO, PL, SE, CH
- HPE, AMD EPYC CPUs and AMD Instinct GPUs
- Peak: 552 Pflop/s

Leonardo

- Hosted at CINECA (IT), partners AT, SK, SI, HU
- Bull Sequana, Intel Xeon CPUs and NVIDIA A100 GPUs
- Peak: ~250 Pflop/s

## Supercomputing landscape

**EuroHPC Pre-exascale systems, coming on-line 2021**

LUMI

- Hosted at CSC (FI), partners BE, CZ, DK, EE, IS, NO, PL, SE, CH
- HPE, AMD EPYC CPUs and AMD Instinct GPUs
- Peak: 552 Pflop/s

Leonardo

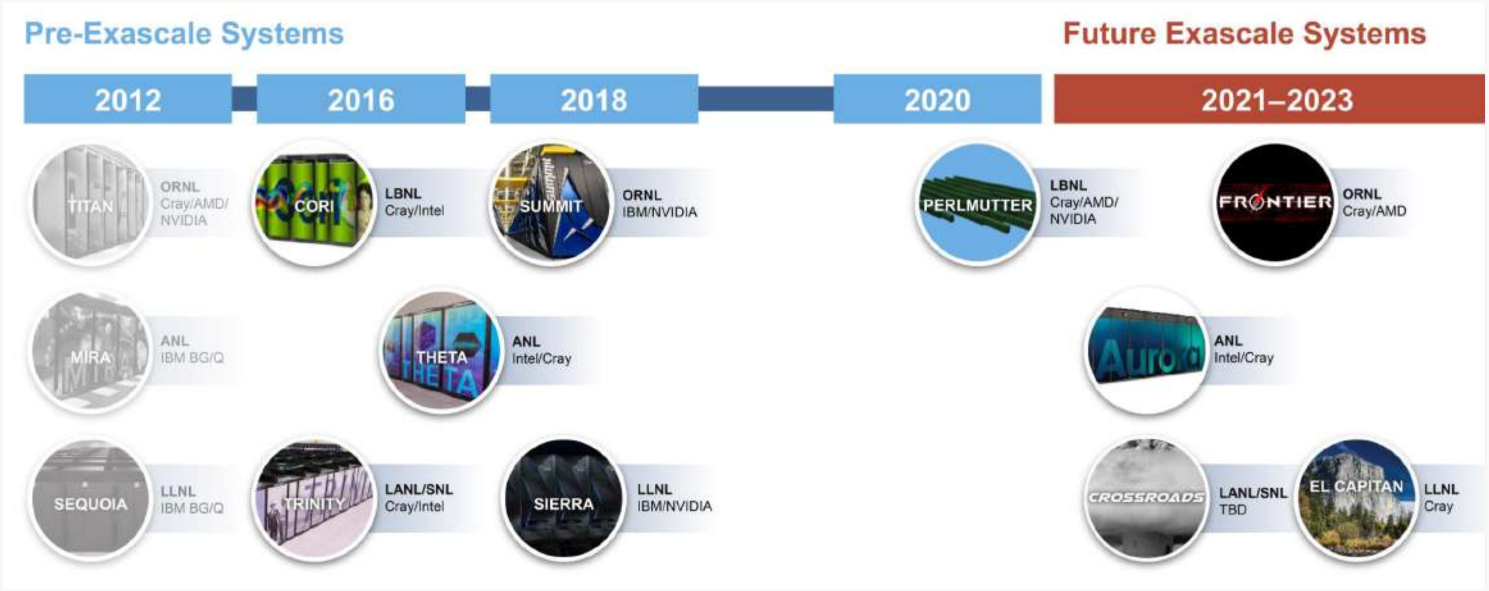
- Hosted at CINECA (IT), partners AT, SK, SI, HU
- Bull Sequana, Intel Xeon CPUs and NVIDIA A100 GPUs
- Peak: $\sim$250 Pflop/s

MareNostrum 5

- Hosted at BSC (ES), partners PR, TR, HR
- Architecture not announced yet other than "same as MareNostrum 4". Intel GPUs?
- Peak: $\sim$200 Pflop/s

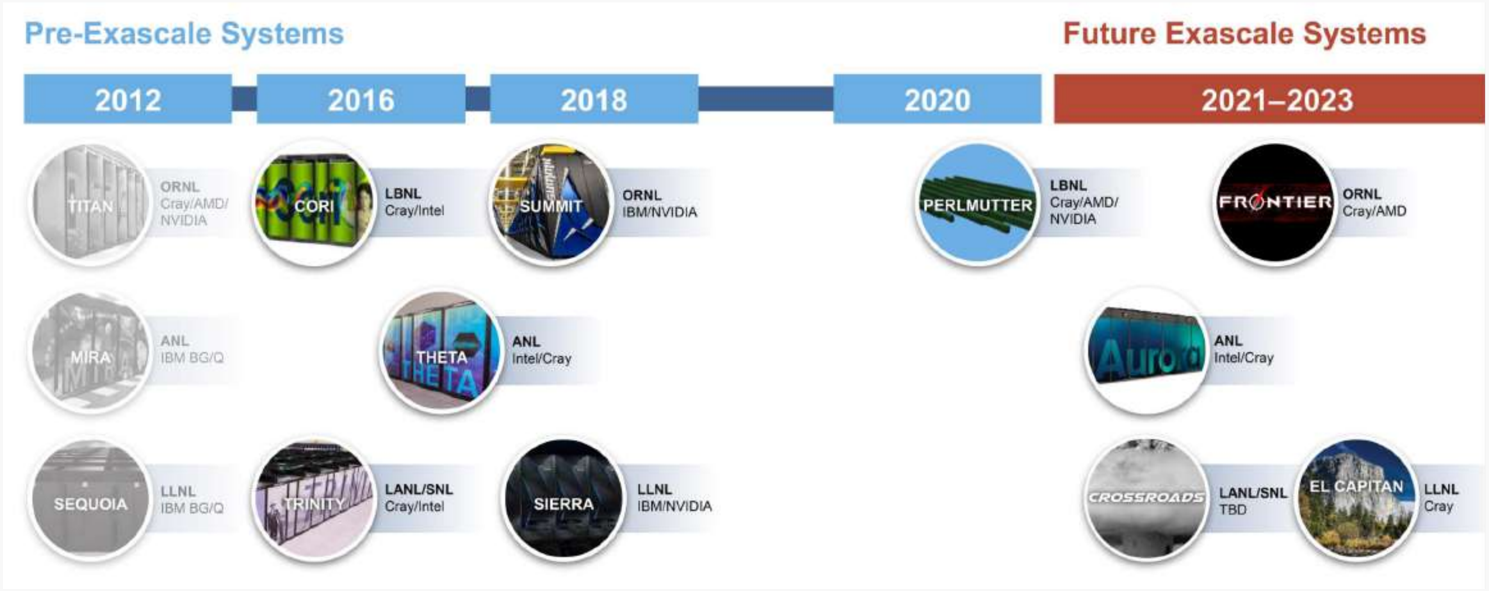
# Supercomputing landscape

## US Exascale roadmap



# Supercomputing landscape

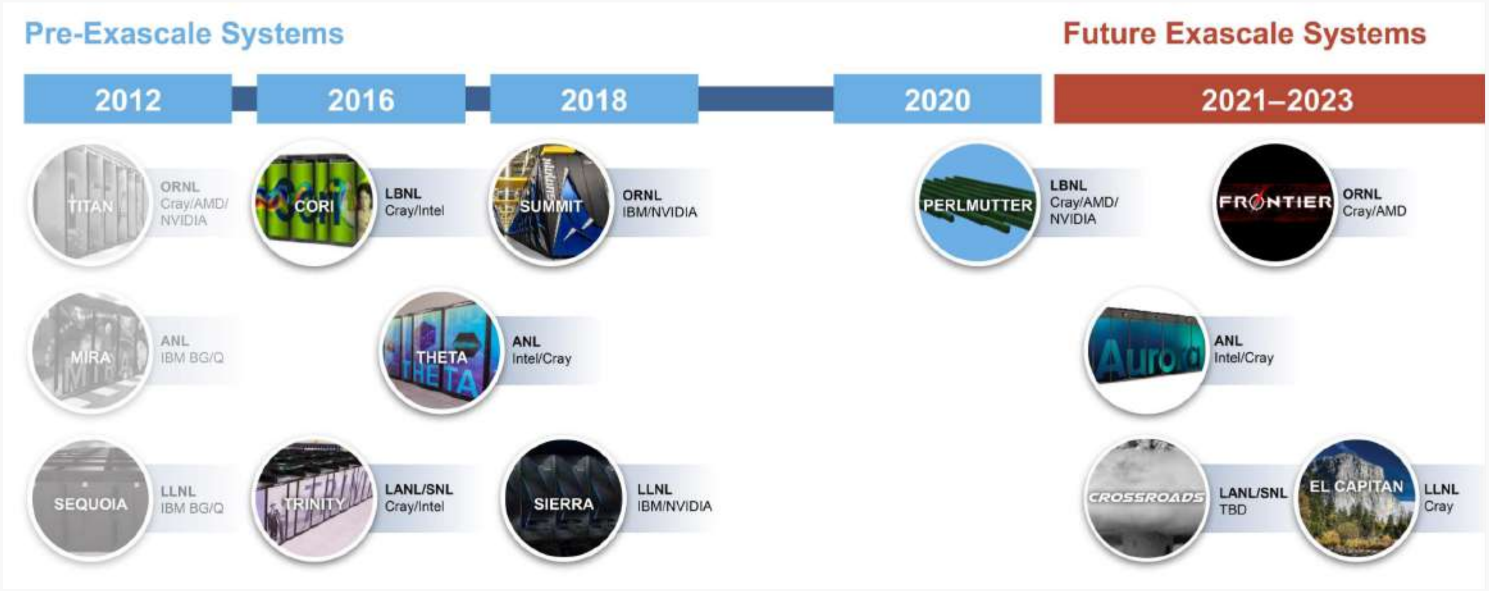
### US Exascale roadmap



- First exascale system "Aurora" expected 2021 with Intel GPUs

## Supercomputing landscape

### US Exascale roadmap



- First exascale system "Aurora" expected 2021 with Intel GPUs
- Similar mix as in Europe: Expect significant performance from NVIDIA, AMD, and Intel GPUs

## Assessing performance

of scientific application kernels

## Performance of scientific codes

**Considerations**

Peak floating point rate

- The theoretical, highest number of floating point operations that can be carried out by a computational unit
- Depends on: clock rate, vector length, FPUs per core, cores per socket

Peak bandwidth

- The theoretical, highest number of bytes that can be read/written from/to some level of memory (L1,2,3 cache, RAM, etc.)
- For RAM: data rate, channels, ranks, banks

## Performance of scientific codes

**Considerations**

Peak floating point rate

- The theoretical, highest number of floating point operations that can be carried out by a computational unit
- Depends on: clock rate, vector length, FPUs per core, cores per socket

Peak bandwidth

- The theoretical, highest number of bytes that can be read/written from/to some level of memory (L1,2,3 cache, RAM, etc.)
- For RAM: data rate, channels, ranks, banks

Good to have these numbers at hand for the machine being used

## Performance of scientific codes

**An Intel Xeon system example (Juwels)**

- Peak FP ($\gamma_{\mathrm{FP}}$)
  - 48 cores per node, clock: 2.7 GHz (2$\times$24-core Intel Skylake)
  - Best case: two 512-bit fused multiply-and-adds per cycle (AVX-512)
- In double precision: 2$\times$(8 `mul` + 8 `add`) per cycle = 32 flop/cycle
  - Therefore: 2.7x$10^9$ cycles/s $\times$ 32 flop/cycle = 86.4 Gflop/s per core
  - 4,147.2 Gflop/s per node
- Peak BW ($\gamma_{\mathrm{IO}}$)
  - 2666 MHz six-channel DDR4: 128 GBytes/s per socket
  - 256 GBytes/s (dual socket)
- Some (semi-)standard tools
  - On Linux, you can obtain processor details via `cat /proc/cpuinfo`.
  - You can obtain topology and memory info e.g. `hwloc`, `dmidecode` (latter requires access to `/dev/mem`)

## Computational kernels

**Sustained performance**

- Sustained FP-rate: the measured, average number of floating point operations carried out by the kernel per unit time
    - `add`, `sub`, and `mul` count as 1 flop
    - `dev`, `sqrt`, `sin`, etc. count $\geqslant$ 2 flops. Depends on architecture
    - Count number of flops in kernel and divide by runtime
    - Alternatively, or for more complex codes, use performance counters

**In our examples we will see cases of kernels where the flops are countable**

- Sustained BW: the measured, average bytes read/written from main memory per unit time
    - As in the case of FP-rate, count bytes needed to be read and bytes needed to be written to and from RAM and divide by run time
    - Maximum data reuse assumption: once data is read from RAM, it never needs to be re-read

## An example

- Consider the following kernel operation:

$$Y_i^{ab} = A^{ac} \cdot X_i^{cb},$$

with: $i = 0, \ldots, L-1$ , $a, b, c = 0, \ldots, N-1$ , $L \gg N$

- Straight forward implementation:

```
double Y[L][N][N];
double X[L][N][N];
double A[N][N];

/* ...
 * Initialize X[][][] and A[][]
 * ...  */
for(int i=0; i<L; i++) {
  for(int a=0; a<N; a++) {
    for(int b=0; b<N; b++) {
      Y[i][a][b] = 0;
      for(int c=0; c<N; c++) {
         Y[i][a][b] += A[a][c]*X[i][c][b];
      }
    }
  }
}
```

**Number of fp operations**

- $N_{FP} = L \cdot 2 \cdot N^3$

**Number of bytes of I/O**

- $N_{IO} = w \cdot (2 \cdot L \cdot N^2 + N^2)$
- $w$: word-length in bytes, e.g. $w = 4$ for single precision, $8$ for double, etc.
- $2LN^2 \rightarrow O(L)$ if $L \gg N$
- In any case, if $N$ small enough, `A` should be kept in low-level cache

## An example

### Data reuse assumption

```
Y[i][0][0] = A[0][0]*X[i][0][0] + A[0][1]*X[i][1][0] + ... + A[0][N-1]*X[i][N-1][0];
Y[i][0][1] = A[0][0]*X[i][0][1] + A[0][1]*X[i][1][1] + ... + A[0][N-1]*X[i][N-1][0];
...
Y[i][1][0] = A[1][0]*X[i][0][0] + A[1][1]*X[i][1][0] + ... + A[1][N-1]*X[i][N-1][0];
Y[i][1][1] = A[1][0]*X[i][0][1] + A[1][1]*X[i][1][1] + ... + A[1][N-1]*X[i][N-1][1];
...
```

Elements of `X` and `A` are required multiple times. However we only count their loads once.

- Given some measurement of the run-time $\bar{T}$
  - FP-rate: $\beta_{FP} = \frac{N_{FP}}{\bar{T}}$
  - IO-rate: $\beta_{IO} = \frac{N_{IO}}{\bar{T}}$
- This motivates defining an *intensity* $I = \frac{N_{FP}}{N_{IO}}$

```
for(int i=0; i<L; i++) {
  for(int a=0; a<N; a++) {
    for(int b=0; b<N; b++) {
      Y[i][a][b] = 0;
      for(int c=0; c<N; c++) {
           Y[i][a][b] += A[a][c]*X[i][c][b];
      }
    }
  }
}
```

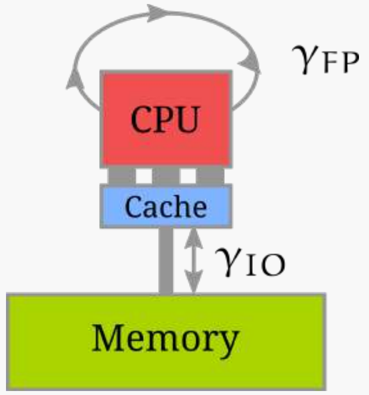
# Intensities

**Computational kernel intensity**

- Ratio of kernel floating point operations to bytes of I/O
- For our previous example:
  - $I_k = \frac{N_{FP}}{N_{IO}} = \frac{2LN^3}{2wLN^2} = N/w$
  - Note how the problem size $L$ drops out $\Rightarrow$ constant $I_k$ irrespective of problem size
  - E.g. for $N = 3$ and double precision, $I_k = 0.375$ flops/byte

**Machine flop/byte ratio**

- Similarly to $I_k$, we can define the machine flop/byte ratio ($I_m$)
- $I_m = \frac{\gamma_{FP}}{\gamma_{IO}}$
- E.g. for Juwels compute node: $I_m \simeq 16.2 \frac{\text{flop}}{\text{byte}}$



# Intensities

**Balance between kernel / hardware intensities**

- $I_k \gg I_m$ : Kernel is "compute-bound" on this architecture. Higher $\gamma_{FP}$ would lead to higher performance, but higher $\gamma_{IO}$ would not.
- $I_k \ll I_m$ : Kernel is "bandwidth-" or "memory-bound" on this architecture. Higher $\gamma_{IO}$ would lead to higher performance, but higher $\gamma_{FP}$ would not.
- $I_k \simeq I_m$ : Kernel is balanced on this architecture. Ideal situation.

**For the example kernel of the previous slides, on our example hardware**

- $I_k \ll I_m \Rightarrow$ the kernel is memory-bound

**Note the assumptions that enter $I_k$ and $I_m$**

- $\gamma_{FP}$ considers all operations can be a sequence of multiply-and-add
- $\beta_{IO}$ assumes maximum data reuse
- $I_k$ constant if problem size $L$ drops out

## Expectations of performance

**Given an architecture with $\mathrm{I}_\mathrm{m}$**

- Can we know $\mathrm{N}_{\mathrm{FP}}$ and $\mathrm{N}_{\mathrm{IO}}$ for the kernel we wish to run; can we estimate $\mathrm{I}_\mathrm{k}$?
- Compare $\mathrm{I}_\mathrm{k}$ and $\mathrm{I}_\mathrm{m}$. Do we expect the kernel to be memory or compute bound on this architecture?
- Can we obtain $\beta_{\mathrm{FP}}$ and $\beta_{\mathrm{IO}}$?
  - For this, we need to measuring the performance on the targeted architecture
- What are the ratios $\frac{\beta_{\mathrm{FP}}}{\gamma_{\mathrm{FP}}}$ and $\frac{\beta_{\mathrm{IO}}}{\gamma_{\mathrm{IO}}}$?

# Expectations of performance

**Given an architecture with $I_m$**

- Can we know $N_{FP}$ and $N_{IO}$ for the kernel we wish to run; can we estimate $I_k$?
- Compare $I_k$ and $I_m$. Do we expect the kernel to be memory or compute bound on this architecture?
- Can we obtain $\beta_{FP}$ and $\beta_{IO}$?
  - For this, we need to measuring the performance on the targeted architecture
- What are the ratios $\frac{\beta_{FP}}{\gamma_{FP}}$ and $\frac{\beta_{IO}}{\gamma_{IO}}$?

**This analysis should guide our overall strategy when developing or optimizing computational kernels**

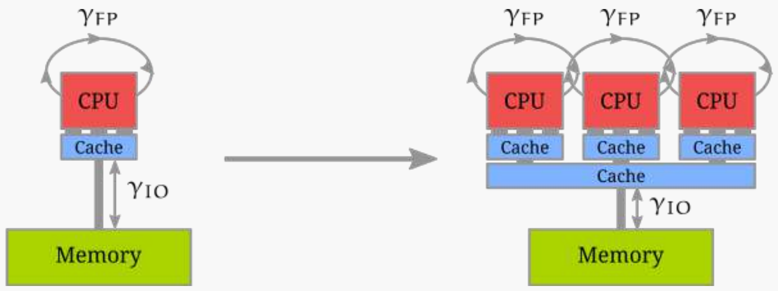
- If the kernel is memory-bound, we should be trying to optimize for memory I/O. Ideally we try to achieve a $\frac{\beta_{IO}}{\gamma_{IO}} \longrightarrow 1$.
- If the kernel is compute-bound, we should be trying to optimize for a higher FP-rate. Ideally we try to achieve a $\frac{\beta_{FP}}{\gamma_{FP}} \longrightarrow 1$.

## Strategies

Some node-level considerations for optimization

## Strategies

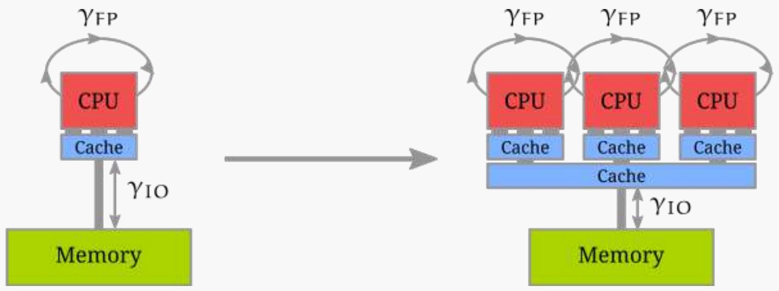
**Mutli-threading**



- For compute-bound kernels, this effectively multiplies $\gamma_{\mathrm{FP}}$
- For memory-bound kernels, allows better saturation of $\gamma_{\mathrm{IO}}$

## Strategies

**Mutli-threading**



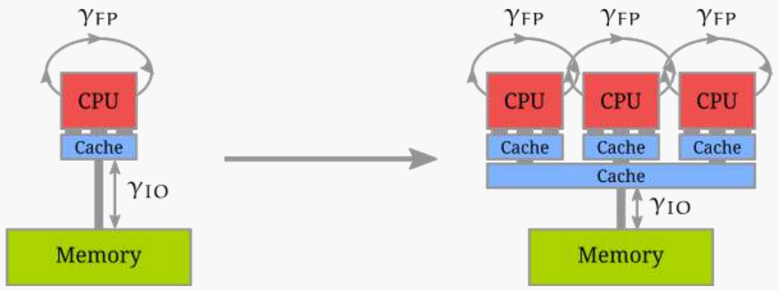
- For compute-bound kernels, this effectively multiplies $\gamma_{\text{FP}}$
- For memory-bound kernels, allows better saturation of $\gamma_{\text{IO}}$

**Vectorization**

- Enable the use of specialized SIMD hardware
- Also benefits efficient I/O

## Strategies

**Mutli-threading**



- For compute-bound kernels, this effectively multiplies $\gamma_{\text{FP}}$
- For memory-bound kernels, allows better saturation of $\gamma_{\text{IO}}$

**Vectorization**

- Enable the use of specialized SIMD hardware
- Also benefits efficient I/O

**Data layout transformations**

- Transformations for mitigation of cache misses (improve temporal and spatial cache locality) $\longrightarrow$ Blocking and tiling
- Transformations which can assist vectorization or auto-vectorization

## Vectorization

**Most modern processors have some vectorization capabilities**

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)
  y[i] = a*x[i] + y[i];
```

## Vectorization

**Most modern processors have some vectorization capabilities**

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)
  y[i] = a*x[i] + y[i];
```

```
for(int i=0; i<L; i+=4) {
  y[i  ] = a*x[i  ] + y[i  ];
  y[i+1] = a*x[i+1] + y[i+1];
  y[i+2] = a*x[i+2] + y[i+2];
  y[i+3] = a*x[i+3] + y[i+3];
}
```

## Vectorization

**Most modern processors have some vectorization capabilities**

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)
  y[i] = a*x[i] + y[i];
```

```
for(int i=0; i<L; i+=4) {
  y[i  ] = a*x[i  ] + y[i  ];
  y[i+1] = a*x[i+1] + y[i+1];
  y[i+2] = a*x[i+2] + y[i+2];
  y[i+3] = a*x[i+3] + y[i+3];
}
```

```
float4 va = {a,a,a,a};
for(int i=0; i<L; i+=4) {
  float4 vy;
  float4 vx;
  load4(vx, &x[i]);
  load4(vy, &y[i]);
  vy = va*vx + vy;
  store4(&y[i], vy);
}
```

**Note: the above is pseudo-code, namely `load4`, `float4`, etc. are simplifications.**

## Vectorization

**Most modern processors have some vectorization capabilities**

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)
  y[i] = a*x[i] + y[i];
```

```
for(int i=0; i<L; i+=4) {
  y[i  ] = a*x[i  ] + y[i  ];
  y[i+1] = a*x[i+1] + y[i+1];
  y[i+2] = a*x[i+2] + y[i+2];
  y[i+3] = a*x[i+3] + y[i+3];
}
```

```
float4 va = {a,a,a,a};
for(int i=0; i<L; i+=4) {
  float4 vy;
  float4 vx;
  load4(vx, &x[i]);
  load4(vy, &y[i]);
  vy = va*vx + vy;
  store4(&y[i], vy);
}
```

- In many case, the compiler will generate vector instructions (auto-vectorization)
- However this usually requires an appropriate data layout

**Note: the above is pseudo-code, namely `load4`, `float4`, etc. are simplifications.**

## Data layout transformations

**Easier vectorization**

- The data are ordered such that the same operation can be applied to consecutive elements
    - Assists the compiler in detecting auto-vectorization opportunities
    - Assists the programmer in using *vector intrinsics*

## Data layout transformations

**Easier vectorization**

- The data are ordered such that the same operation can be applied to consecutive elements
    - Assists the compiler in detecting auto-vectorization opportunities
    - Assists the programmer in using *vector intrinsics*

**Better cache locality**

- The data are ordered in a way as close to the order in which they will be accessed in your kernels as possible
- The data are reordered such that when an element needs to be accessed multiple times, these multiple accesses are issued very close to each other
- Optimizes for *temporal* and for *spatial* locality

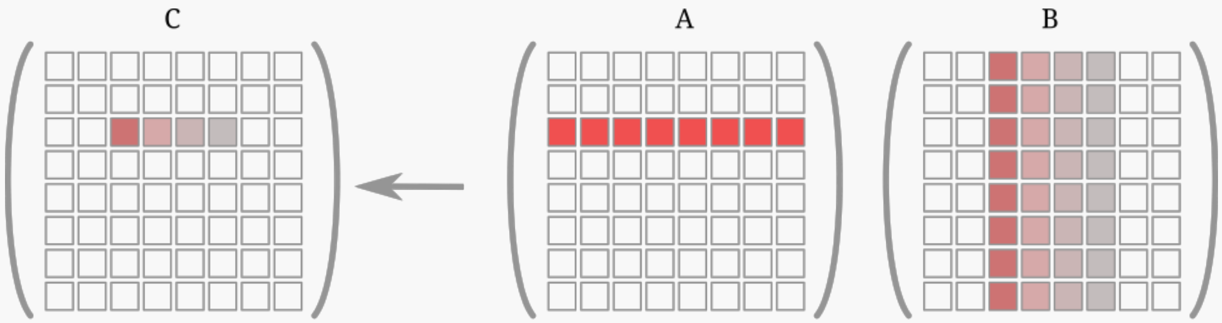
# Data layout transformations

**Example: Matrix-matrix multiplication**

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj}$$

$$A_{M\times N}, B_{N\times M}, C_{M\times M}$$

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



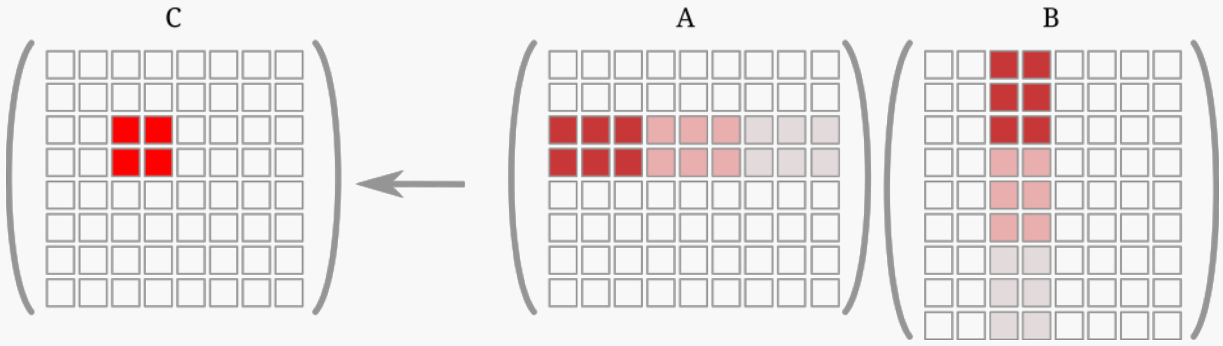
- Repeated reading of columns of $B$
- If $N$ too large, a whole row of $A$ may not fit into L2 cache

## Data layout transformations

**Example: Matrix-matrix multiplication**

Blocking: better *cache locality*

```
for(int i=0; i<M; i+=BM)
  for(int j=0; j<M; j+=BM) {
    Cb[:BM][:BM] = 0;
    for(int k=0; k<N; k++) {
      Ab[:BM][:BN] = A[i:i+BM][k:k+BN];
      Bb[:BN][:BM] = B[k:k+BN][j:j+BM];
      for(int ib=0; ib<BM; ib++)
        for(int jb=0; jb<BM; jb++)
          for(int kb=0; kb<BN; kb++)
            Cb[ib][jb] += Ab[ib][kb] * Bb[kb][jb];
      C[i:i+BM][j:j+BM] = Cb[:BM][:BM];
    }
  }
```

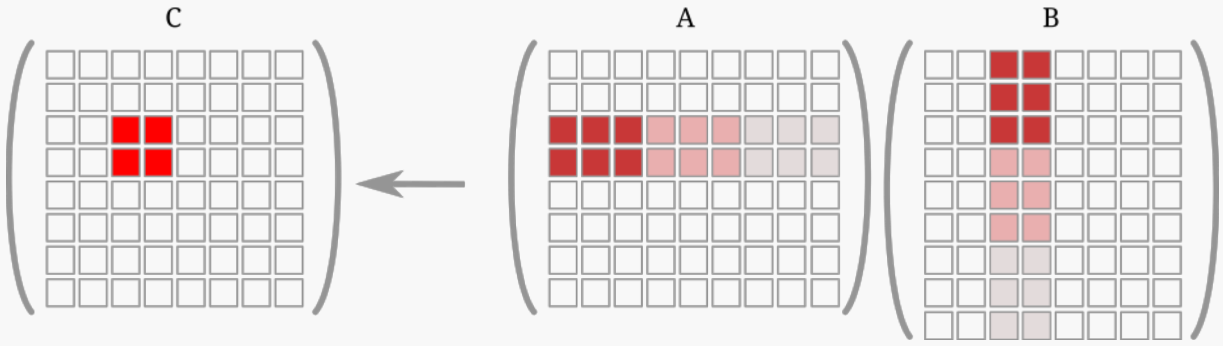


## Data layout transformations

**Example: Matrix-matrix multiplication**

Transpose tiles of $B$: better *alignment* for SIMD

```
for(int i=0; i<M; i+=BM)
  for(int j=0; j<M; j+=BM) {
    Cb[:BM][:BM] = 0;
    for(int k=0; k<N; k++) {
      Ab[:BM][:BN] = A[i:i+BM][k:k+BN];
      Bb[:BM][:BN] = B[k:k+BN][j:j+BM];              \* Transpose B *\
      for(int ib=0; ib<BM; ib++)
        for(int jb=0; jb<BM; jb++)
          for(int kb=0; kb<BN; kb++)
            Cb[ib][jb] += Ab[ib][kb] * Bb[jb][kb]; \* Transpose B *\
      C[i:i+BM][j:j+BM] = Cb[:BM][:BM];
    }
  }
```



## Data layout transformations

**AoS: Array of structures**

Arrays of structures arise naturally when mapping physical problems to code. E.g. say you want to define an array of coordinates:

```
typedef struct {
  float x;
  float y;
  float z;
} coords;
```

You can now allocate an array of `coords`:

```
size_t L = 1000;
coords *arr = malloc(sizeof(coords)*L);
```

# Data layout transformations

**AoS: Array of structures**

Arrays of structures arise naturally when mapping physical problems to code. E.g. say you want to define an array of coordinates:

```
typedef struct {
  float x;
  float y;
  float z;
} coords;
```

You can now allocate an array of `coords`:

```
size_t L = 1000;
coords *arr = malloc(sizeof(coords)*L);
```

Now assume you would like to get an array of the distances of the coordinates from the origin:

```
for(int i=0; i<L; i++)
  r[i] = sqrt(arr[i].x*arr[i].x +
              arr[i].y*arr[i].y +
              arr[i].z*arr[i].z);
```

You can see how the choice of data layout makes auto-vectorization difficult. A common way around this is to use a Structure of Arrays (SoA) rather than an Array of Structures.

# Data layout transformations

## AoS to SoA

Define a structure of arrays, this is similar to how one programs for GPUs:

```
typedef struct {
  float *x;
  float *y;
  float *z;
} coords;
```

And now allocate each element of `coords` separately:

```
coords arr;
arr.x = malloc(sizeof(float)*L);
arr.y = malloc(sizeof(float)*L);
arr.z = malloc(sizeof(float)*L);
```

This layout facilitates vectorization

```
for(int i=0; i<L; i+=4) {
  r[i  ] = sqrt(arr.x[i  ]*arr.x[i  ]+arr.y[i  ]*arr.y[i  ]+arr.z[i  ]*arr.z[i  ]);
  r[i+1] = sqrt(arr.x[i+1]*arr.x[i+1]+arr.y[i+1]*arr.y[i+1]+arr.z[i+1]*arr.z[i+1]);
  r[i+2] = sqrt(arr.x[i+2]*arr.x[i+2]+arr.y[i+2]*arr.y[i+2]+arr.z[i+2]*arr.z[i+2]);
  r[i+3] = sqrt(arr.x[i+3]*arr.x[i+3]+arr.y[i+3]*arr.y[i+3]+arr.z[i+3]*arr.z[i+3]);
}
```

## Data layout transformations

**Stencil codes**

Another application of the data re-ordering is in stencil codes. Consider a 2-D stencils operation:

$$\phi_{x,y} \leftarrow 4\psi_{x,y} - (\psi_{x+1,y} + \psi_{x-1,y} + \psi_{x,y+1} + \psi_{x,y-1})$$

```
for(int y=0; y<L; y++)
for(int x=0; x<L; x++) {
 B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);
}
```

Consider an unrolled implementation of this, with `x` running fastest and `y` slowest:

```
for(int y=0; y<L; y++)
for(int x=0; x<L; x+=4) {
 B[y][x  ] = 4*A[y][x  ] - (A[y][x+1] + A[y][x-1] + A[y+1][x  ] + A[y-1][x  ]);
 B[y][x+1] = 4*A[y][x+1] - (A[y][x+2] + A[y][x  ] + A[y+1][x+1] + A[y-1][x+1]);
 B[y][x+2] = 4*A[y][x+2] - (A[y][x+3] + A[y][x+1] + A[y+1][x+2] + A[y-1][x+2]);
 B[y][x+3] = 4*A[y][x+3] - (A[y][x+4] + A[y][x+2] + A[y+1][x+3] + A[y-1][x+3]);
}
```

`A[y][x:x+4]` does not align over all operations

# Data layout transformations

**Stencil codes**

```
for(int y=0; y<L; y++)
for(int x=0; x<L; x+=4) {
 B[y][x  ] = 4*A[y][x  ] - (A[y][x+1] + A[y][x-1] + A[y+1][x  ] + A[y-1][x  ]);
 B[y][x+1] = 4*A[y][x+1] - (A[y][x+2] + A[y][x  ] + A[y+1][x+1] + A[y-1][x+1]);
 B[y][x+2] = 4*A[y][x+2] - (A[y][x+3] + A[y][x+1] + A[y+1][x+2] + A[y-1][x+2]);
 B[y][x+3] = 4*A[y][x+3] - (A[y][x+4] + A[y][x+2] + A[y+1][x+3] + A[y-1][x+3]);
}
```

An alternative is to change the data layout so that we vectorize over distant elements. I.e. we can reshape the data layout:

from: `A[y][x] <- A + y*L + x`
to: `vA[y0][x][y1] = A[y0+y1*L/4][x]`, and re-write the kernel:

```
for(int y=0; y<L/4; y++)
for(int x=0; x<L; x++) {
 vB[y][x][0] = 4*vA[y][x][0] - (vA[y][x+1][0] + vA[y][x-1][0] + vA[y+1][x][0] + vA[y-1][x][0]);
 vB[y][x][1] = 4*vA[y][x][1] - (vA[y][x+1][1] + vA[y][x-1][1] + vA[y+1][x][1] + vA[y-1][x][1]);
 vB[y][x][2] = 4*vA[y][x][2] - (vA[y][x+1][2] + vA[y][x-1][2] + vA[y+1][x][2] + vA[y-1][x][2]);
 vB[y][x][3] = 4*vA[y][x][3] - (vA[y][x+1][3] + vA[y][x-1][3] + vA[y+1][x][3] + vA[y-1][x][3]);
}
```

## Data layout transformations

**Stencil codes**

Shuffling of the elements is still required but restricted to the boundaries. E.g. for periodic boundary conditions, the original code at the `y=0` boundary would be:

```
y = 0;
for(int x=0; x<L; x++) {
 B[0][x] = 4*A[0][x] - (A[0][x+1] + A[0][x-1] + A[1][x] + A[L-1][x]);
}
```

which requires shuffling of the boundary element at `y=L/4-1` of `vA`:

```
y = 0;
for(int x=0; x<L; x++) {
 vB[0][x][0] = 4*vA[0][x][0] - (vA[0][x+1][0] + vA[0][x-1][0] + vA[1][x][0] + vA[L/4-1][x][3]);
 vB[0][x][1] = 4*vA[0][x][1] - (vA[0][x+1][1] + vA[0][x-1][1] + vA[1][x][1] + vA[L/4-1][x][2]);
 vB[0][x][2] = 4*vA[0][x][2] - (vA[0][x+1][2] + vA[0][x-1][2] + vA[1][x][2] + vA[L/4-1][x][1]);
 vB[0][x][3] = 4*vA[0][x][3] - (vA[0][x+1][3] + vA[0][x-1][3] + vA[1][x][3] + vA[L/4-1][x][0]);
}
```

## Practical

Two example kernels to exercise the concepts learned

# Practical: kernel computational intensities

## Matrix-matrix multiplication

- Consider a matrix-matrix multiplication: $C_{M \times K} = A_{M \times N} \cdot B_{N \times K}$

```
double C[M][K];
double A[M][N];
double B[N][K];

for(int m=0; m<M; m++) {
  for(int k=0; k<K; k++) {
    C[m][k] = 0;
    for(int n=0; n<N; n++) {
      C[m][k] += A[m][n]*B[n][k];
    }
  }
}
```

- Compute $N_{FP}$ and $N_{IO}$ as a function of the matrices' dimensions
- Identify when it is compute- and when is it memory-bound on, e.g. Juwels

## Practical: kernel computational intensities

**Complex linear algebra**

- Consider the operation: $y_i = ax_i + y_i,\ i = 1, \ldots, N$ , where the scalar $a$ and the vectors $x$ and $y$ are complex

```
_Complex double x[N];
_Complex double y[N];
_Complex double a;
for(int i=0; i<N; i++) {
  y[i] = a*x[i] + y[i];
}
```

- Compute $N_{FP}$ and $N_{IO}$ as a function of the matrices' dimensions
- Identify when it is compute- and when is it memory-bound on, e.g. Juwels