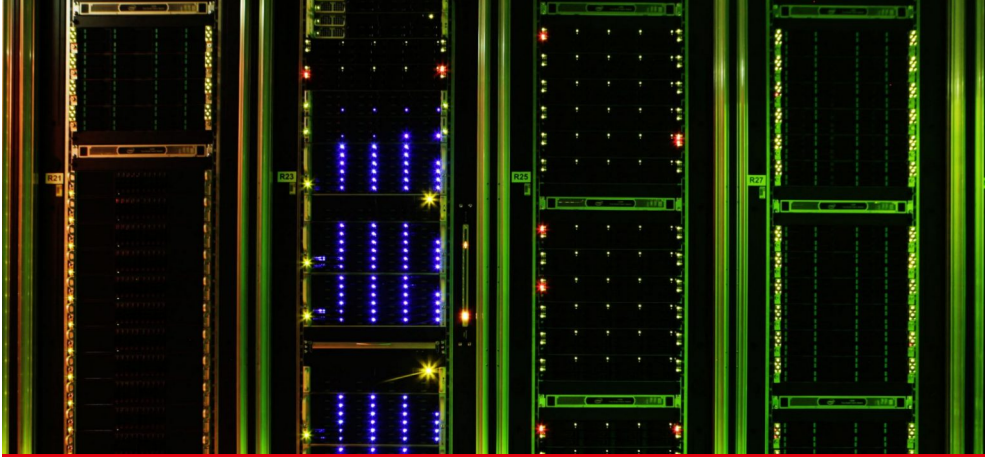


MATH-454 Parallel and High Performance Computing

Lecture 2: Performance and single-core optimization

Pablo Antolin

Slides of N. Richart, E. Lanti, V. Keller's lecture notes

March 6 2025

Performance measurement

Goal of this section

- Key concepts to quantify performance
  - Metrics
  - Scalings, speedup, efficiency
- Roofline model

Performance metrics

- How can we quantify performance?
- We need to define a way of measuring it
- We will focus on the most common metrics for HPC

Performance metrics

- How can we quantify performance?
- We need to define a way of measuring it
- We will focus on the most common metrics for HPC

- The first that comes in mind is *time*, e.g., time-to-solution
- Derived metrics: speedup and efficiency

Performance metrics

- How can we quantify performance?
- We need to define a way of measuring it
- We will focus on the most common metrics for HPC

- The first that comes in mind is *time*, e.g., time-to-solution
- Derived metrics: speedup and efficiency

- Scientific codes do computations on floating point numbers
- A second metric is the number of *floating-point operations per second* (FLOP/s)

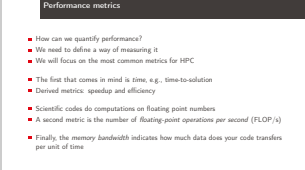
Performance metrics

- How can we quantify performance?
- We need to define a way of measuring it
- We will focus on the most common metrics for HPC

- The first that comes in mind is *time*, e.g., time-to-solution
- Derived metrics: speedup and efficiency

- Scientific codes do computations on floating point numbers
- A second metric is the number of *floating-point operations per second* (FLOP/s)

- Finally, the *memory bandwidth* indicates how much data does your code transfers per unit of time



- My code is super fast, it runs in 2.5ns!
- It seems fast, but is it? How fast your hardware is?
- To really understand how much your code exploits the hardware, we use the FLOP/s and memory bandwidth (B/s)
- Your hardware has theoretical maximum values for those
- You can compare the values from your code to the max to see how well you use the hardware

Speedup and efficiency

- Two important metrics are derived from timings
- Compare timings with p processes, T_p, against the reference timing, T_1

Speedup

$$S(p) = \frac{T_1}{T_p}$$

Efficiency

$$E(p) = \frac{S(p)}{p}$$

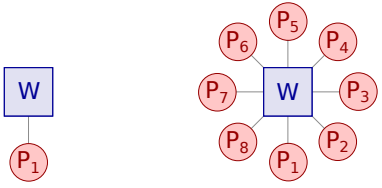
- We want $S(p)$ as close to p and $E(p)$ as close to 1 (100%) as possible

Strong scaling

- Scalings are a way to assess how well a program performs when adding computational resources
- Strong scaling: add resources, keep total amount of work constant

$$S(p) = \frac{T_1}{T_p}, \qquad E(p) = \frac{S(p)}{p} = \frac{T_1}{pT_p}$$

- Strong scaling is an indication on how much profitable it is to add resources to solve your problem

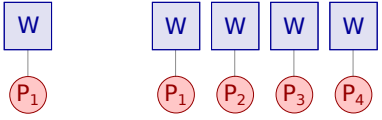


Weak scaling

- Weak scaling: add resources and maintain amount of work per resource constant

$$S(p) = \frac{pT_1}{T_p}, \qquad E(p) = \frac{S(p)}{p} = \frac{T_1}{T_p}$$

- Weak scalings are an indication on how well your code will perform on a bigger machine (and with a bigger problem)
- These scalings are always required for project proposals
  - For strong scalings the metric is speedup (how do I improve performance)
  - For weak scalings the metric is efficiency (how well performance is kept)

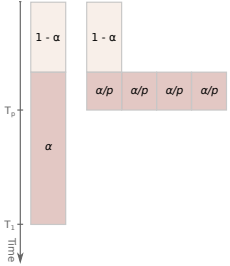


Amdahl's law

- Amdahl's law provides an upper bound to the achievable speedup for a **fixed problem size**
- By definition it is a **strong scaling** analysis

Amdahl's law

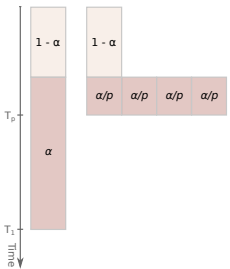
- Amdahl's law provides an upper bound to the achievable speedup for a **fixed problem size**
- By definition it is a **strong scaling** analysis
- Assume a fraction α of your code is (perfectly) parallel and let be T_1 the time with 1 process: $T_1 = (1-\alpha)T_1 + \alpha T_1$
- Time for p processes would be $T_p = (1-\alpha)T_1 + \frac{\alpha}{p}T_1 = \left[(1-\alpha) + \frac{\alpha}{p}\right] T_1$



Amdahl's law

- Amdahl's law provides an upper bound to the achievable speedup for a **fixed problem size**
- By definition it is a **strong scaling** analysis
- Assume a fraction α of your code is (perfectly) parallel and let be T_1 the time with 1 process: $T_1 = (1-\alpha)T_1 + \alpha T_1$
- Time for p processes would be $T_p = (1-\alpha)T_1 + \frac{\alpha}{p}T_1 = \left[(1-\alpha) + \frac{\alpha}{p}\right] T_1$
- Speedup becomes

$$S(p) = \frac{T_1}{T_p} = \frac{1}{(1-\alpha) + \frac{\alpha}{p}}$$



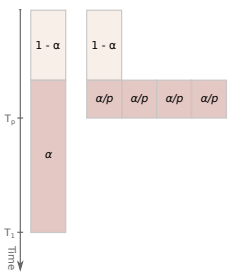
Amdahl's law

- Amdahl's law provides an upper bound to the achievable speedup for a **fixed problem size**
- By definition it is a **strong scaling** analysis
- Assume a fraction α of your code is (perfectly) parallel and let be T_1 the time with 1 process: $T_1 = (1-\alpha)T_1 + \alpha T_1$
- Time for p processes would be $T_p = (1-\alpha)T_1 + \frac{\alpha}{p}T_1 = \left[(1-\alpha) + \frac{\alpha}{p}\right] T_1$
- Speedup becomes

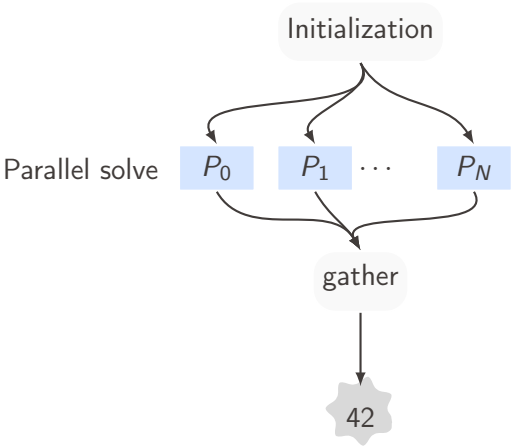
$$S(p) = \frac{T_1}{T_p} = \frac{1}{(1-\alpha) + \frac{\alpha}{p}}$$

- In the limit (with infinite computational resources)

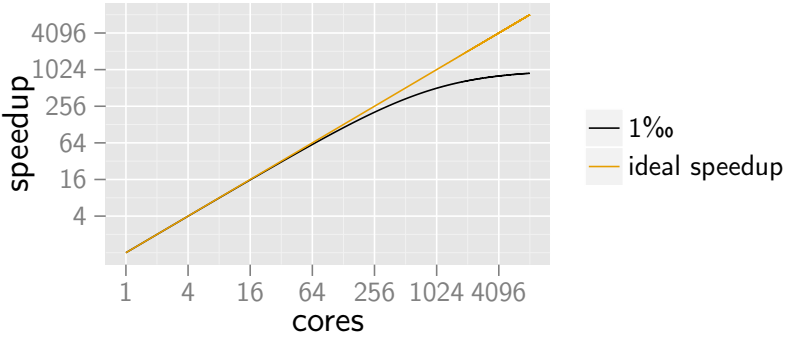
$$\lim_{p\to\infty} S(p) = \frac{1}{1-\alpha}$$



Amdahl's trap: are we doomed?

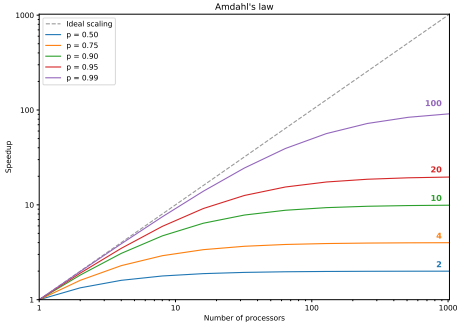


Amdahl's trap: are we doomed?



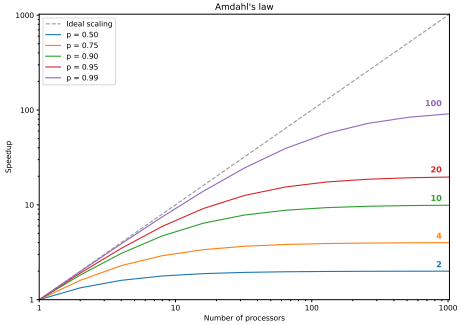
Even if initialization and gathering took up 1‰ of the time, running on *El Capitan* would be a pity

Amdahl's trap: are we doomed?



- Limited by the serial part (very sensitive)!
- Does this mean we cannot exploit large HPC machines?

Amdahl's trap: are we doomed?



- Limited by the serial part (very sensitive)!
- Does this mean we cannot exploit large HPC machines?
- No, in general with more resources, we simulate larger systems $\Rightarrow$ weak scaling (see Gustafson's law)

Amdahl's trap: are we doomed?

Observation:
Amdahl's law is an upper bound for a fixed problem of size N.

Problem:
If we increase the problem size, the relative sequential part can be made smaller and smaller if it is of lower complexity than the parallel part of the problem, i.e., $\lim_{N\to\infty} \alpha(N) = 1$.

Alternative:
Take the problem size N into account. This is Gustafson's law (or Gustafson's complement to Amdahl's law)

Gustafson's Law – weak or soft scaling

- Different perspective compared to Amdahl's law
- Based on **variable workload** instead of fixed problem size

Given p the number of processors, $\alpha(N)$ the fraction of parallelizable code, and N the problem size, the speedup is:

$$S(N, p) = 1 + (p - 1)\alpha(N)$$

Gustafson's Law – weak or soft scaling

- Different perspective compared to Amdahl's law
- Based on **variable workload** instead of fixed problem size

Given p the number of processors, $\alpha(N)$ the fraction of parallelizable code, and N the problem size, the speedup is:

$$S(N,p) = 1 + (p-1)\alpha(N)$$

Explanation:
The parallel execution time on p processors is

$$T_p = \alpha(N)\,T_p + (1-\alpha(N))\,T_p$$

On a sequential machine,

$$T_1 = p\alpha(N)\,T_p + (1-\alpha(N))\,T_p$$

Gustafson's Law – weak or soft scaling

- Different perspective compared to Amdahl's law
- Based on **variable workload** instead of fixed problem size

Given p the number of processors, $\alpha(N)$ the fraction of parallelizable code, and N the problem size, the speedup is:

$$S(N,p) = 1 + (p-1)\alpha(N)$$

Explanation:
The parallel execution time on p processors is

$$T_p = \alpha(N)\,T_p + (1-\alpha(N))\,T_p$$

On a sequential machine,

$$T_1 = p\alpha(N)\,T_p + (1-\alpha(N))\,T_p$$

$$\Rightarrow S(N,p) = T_1/T_p$$

Gustafson's Law – weak or soft scaling

- Different perspective compared to Amdahl's law
- Based on **variable workload** instead of fixed problem size

Given p the number of processors, $\alpha(N)$ the fraction of parallelizable code, and N the problem size, the speedup is:

$$S(N,p) = 1 + (p-1)\alpha(N)$$

As $N \to \infty$, we expect $\alpha(N) \to 1$ and thus $S(N,p) \to p$.

Gustafson's law limitations

- Some problems just do not have large datasets
- In some problems, communication overheads are not negligible
- A combination of the above two: domain decomposition with a small domain
- Sometimes execution time is not proportional to problem size: what if execution time is $O(n^3)$?

Open questions for the audience

Open questions for the audience

... and hints for a course project

Open questions for the audience

... and hints for a course project

- What programs are unsuited for large parallel machines (and must thus be immediately rethought)?

Open questions for the audience

... and hints for a course project

- What programs are unsuited for large parallel machines (and must thus be immediately rethought)?
- How soon do you see it?

Open questions for the audience

... and hints for a course project

- What programs are unsuited for large parallel machines (and must thus be immediately rethought)?
- How soon do you see it?
- What algorithms are suited for big machines?

Open questions for the audience

... and hints for a course project

- What programs are unsuited for large parallel machines (and must thus be immediately rethought)?
- How soon do you see it?
- What algorithms are suited for big machines?
- Gustafson's law is limited by how big our problem can be, while Amdahl's law's limits show up with a high number of processors. How can we maximize our efficiency taking account of this?

FLOP/s and memory bandwidth

- FLOPs are floating point operations, e.g., $+, -, \times, \div$
- Can be evaluated by hand, dividing the number of operations by the running time

- Memory bandwidth measures the amount of data transferred by unit of time [B/s, KiB/s, MiB/s, GiB/s, ...]
- Can be measured by hand dividing the amount of data transferred by the running time

- In both cases, generally use tools such as PAPI, Tau, likwid, Intel Amplxe, STREAM, etc.

- Assume Intel Xeon Gold 6132 (Gacrux)

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- My code runs in 174.25 ms. It is amazingly fast!

- Assume Intel Xeon Gold 6132 (Gacrux)

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- My code runs in 174.25 ms. It is amazingly fast!
- Each iteration has 2 FLOP (1 add and 1 mul) and there are `N = 1e8` iterations
- Our code $2 \cdot 10^8$ FLOP$/174.25 \cdot 10^{-3}$ s $= 0.001$ TFLOP/s...
- Our hardware can achieve a theoretical peak performance of 1.16 TFLOP/s

- Assume Intel Xeon Gold 6132 (Gacrux)

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- My code runs in 174.25 ms. It is amazingly fast!
- Each iteration has 2 FLOP (1 add and 1 mul) and there are `N = 1e8` iterations
- Our code $2 \cdot 10^8$ FLOP$/174.25 \cdot 10^{-3}$ s $= 0.001$ TFLOP/s...
- Our hardware can achieve a theoretical peak performance of 1.16 TFLOP/s
- Each iteration has 3 memory operations (2 loads and 1 store)
- Our code 2.23 GiB$/174.25 \cdot 10^{-3}$ s $= 12.82$ GiB/s...
- Our hardware can achieve a theoretical memory bandwidth of 125 GiB/s

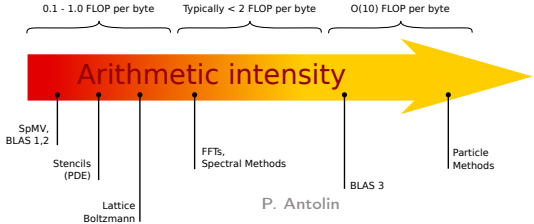
Roofline model

- How well am I exploiting the hardware resources?
- The roofline model is a performance model allowing to have an estimate to this question

Roofline model

- How well am I exploiting the hardware resources?
- The roofline model is a performance model allowing to have an estimate to this question

- Key concept: the arithmetic intensity, AI, of an algorithm is # FLOP/B of data transferred
- It measures data reuse



Roofline model
How to find arithmetic intensity

- For very simple algorithms, you can compute the AI
- Let's take back the DAXPY example

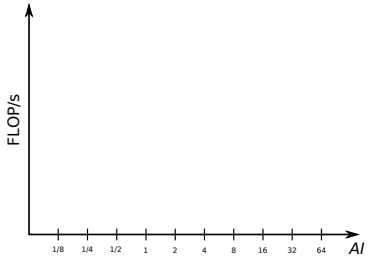
optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- There are 2 operations (1 add and 1 mul)
- Three 8 B memory operations (2 loads and 1 store)
- The AI is then $2/24\,\text{FLOP/B} = 0.083\,\text{FLOP/B}$

Roofline model

- Roofline model is plotted on **log-log scale**
  - x-axis is the *AI*
  - y-axis is FLOP/s

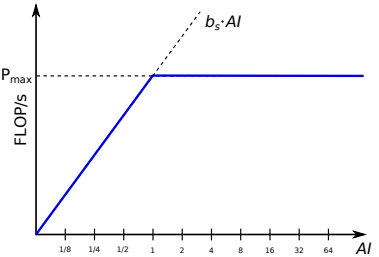


Roofline model

- Roofline model is plotted on **log-log scale**
  - x-axis is the AI
  - y-axis is FLOP/s
- The hardware limits are defined by

$$P = \min(P_{\text{max}}, b_s \cdot AI)$$

  - P_{max} is the CPU peak FLOP/s
  - AI is the intensity
  - b_s is the memory BW

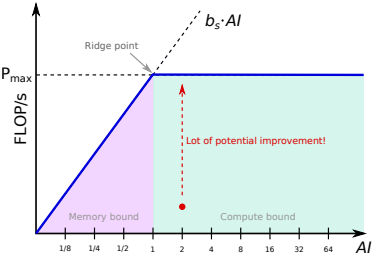


Roofline model

- Roofline model is plotted on **log-log scale**
  - x-axis is the AI
  - y-axis is FLOP/s
- The hardware limits are defined by

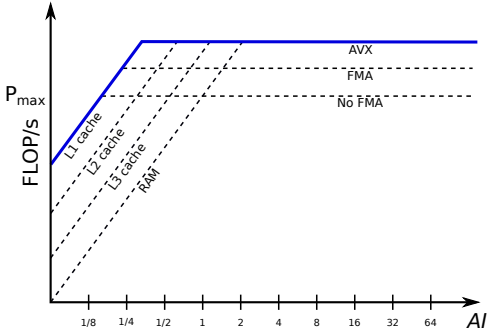
$$P = \min(P_{\text{max}}, b_s \cdot AI)$$

  - P_{max} is the CPU peak FLOP/s
  - AI is the intensity
  - b_s is the memory BW



Roofline model

- Refinements can be made to the Roofline model
- Adding a memory hierarchy with caches
- Adding different levels of DLP (Data-Level parallelism)
- They give you hint on what to optimize for

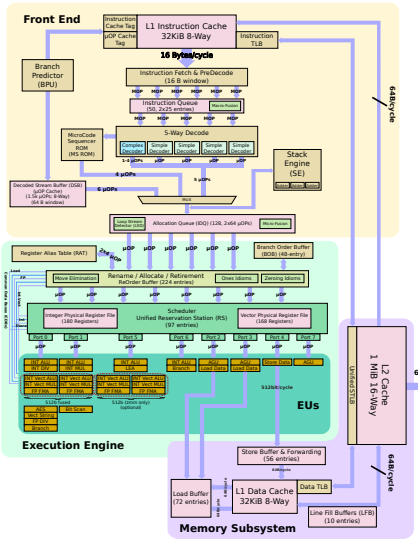


- Theoretical peak performance

  $P_{\max} =$ Number of FP ports (ILP)
  $\times$ flops/cycles (e.g., 2 for FMA)
  $\times$ vector size (DLP) $\times$ frequency (in GHz)
  $\times$ number of cores (TLP)

- Example: Intel Xeon Gold 6132

  $P_{\max} = 2\ \text{(ports)} \times 2\,\text{FLOP/c (2 for FMA)}$
  $\times \dfrac{512\,\text{bit (AVX512)}}{64\,\text{bit (double)}} \times 2.3\,\text{GHz} \times 14\ \text{cores}$
  $= 1.16\,\text{TFLOP/s}$



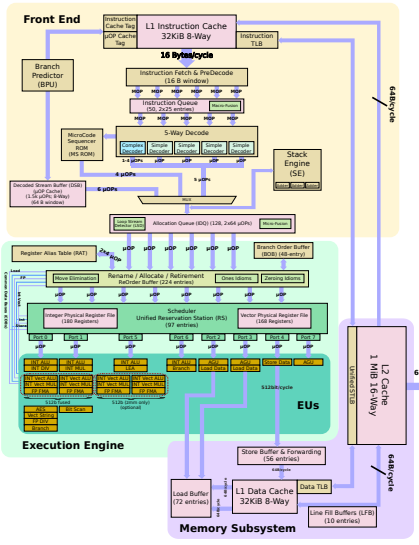
- Theoretical peak performance

  $P_{\max} =$ Number of FP ports (ILP)
  $\times$ flops/cycles (e.g., 2 for FMA)
  $\times$ vector size (DLP) $\times$ frequency (in GHz)
  $\times$ number of cores (TLP)

- Example: Intel Xeon Gold 6132

  $P_{\max} = 2\ \text{(ports)} \times 2\,\text{FLOP/c (2 for FMA)}$
  $\times \dfrac{512\,\text{bit (AVX512)}}{64\,\text{bit (double)}} \times 2.3\,\text{GHz} \times 14\ \text{cores}$
  $= 1.16\,\text{TFLOP/s}$

- Or use a software that estimates it



- Theoretical memory bandwidth of the memory

$$\text{BW}_{\text{max}} = \text{Number of transfers per second} \times \text{Bus width} \times \text{Number of channels}$$

- We assume that RAM matches CPU bandwidth (found on the CPU spec. list)
- Example: Intel Xeon Gold 6132

$$\text{BW}_{\text{max}} = 2666\,\text{MT/s (DDR4 2666)} \times 8\,\text{B/T (64bit bus)} \times 6\text{ (channels)}$$

  - 19.86 GiB/s for 1 channel
  - Maximum of 119.18 GiB/s
  - Ridge point: 1.16 TFLOP/s/119.16 GiB/s = 9.07 FLOP/B
  - Ridge point 1 core: 82.9 GFLOP/s/19.86 GiB/s = 3.89 FLOP/B

Roofline model

How to find the memory bandwidth

- Theoretical memory bandwidth of the memory

$$\text{BW}_{\text{max}} = \text{Number of transfers per second} \times \text{Bus width} \times \text{Number of channels}$$

- We assume that RAM matches CPU bandwidth (found on the CPU spec. list)
- Example: Intel Xeon Gold 6132

$$\text{BW}_{\text{max}} = 2666\,\text{MT/s (DDR4 2666)} \times 8\,\text{B/T (64bit bus)} \times 6\text{ (channels)}$$

  - 19.86 GiB/s for 1 channel
  - Maximum of 119.18 GiB/s
  - Ridge point: 1.16 TFLOP/s/119.16 GiB/s = 9.07 FLOP/B
  - Ridge point 1 core: 82.9 GFLOP/s/19.86 GiB/s = 3.89 FLOP/B
- Or use a software that estimates it
- A corollary for "theoretical" is that it is not achievable in practice!

Roofline model
How to find arithmetic intensity

- For very simple algorithms, you can compute the AI
- Let's take back the DAXPY example

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- There are 2 operations (1 add and 1 mul)
- Three 8 B memory operations (2 loads and 1 store)
- The AI is then $3/24\,\text{FLOP/B} = 0.083\,\text{FLOP/B}$

Roofline model
How to find arithmetic intensity

- For very simple algorithms, you can compute the AI
- Let's take back the DAXPY example

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- There are 2 operations (1 add and 1 mul)
- Three 8 B memory operations (2 loads and 1 store)
- The AI is then $3/24\,\text{FLOP/B} = 0.083\,\text{FLOP/B}$
  $\ll 9.07\,\text{FLOP/B} \Rightarrow$ memory bounded

Roofline model
How to find arithmetic intensity

- For very simple algorithms, you can compute the AI
- Let's take back the DAXPY example

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

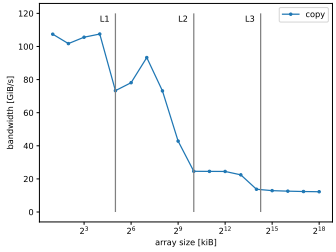
- There are 2 operations (1 add and 1 mul)
- Three 8 B memory operations (2 loads and 1 store)
- The AI is then $3/24\,\text{FLOP/B} = 0.083\,\text{FLOP/B}$

- For more complex algorithms, use a tool, e.g., Intel Advisor

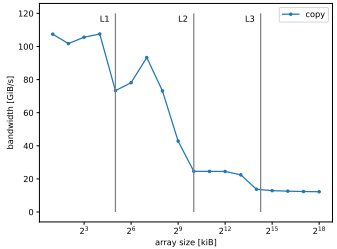
Roofline model

How to measure the actual values

- Peak performance measurement
  - Using a compute bound kernel
  - Using dgemm:
    1 core: 98.0 GFLOP/s
    14 cores: 965.0 GFLOP/s
- Bandwidth measurement
  - Using a memory bound kernel
  - Using stream (triad):
    1 core: 12.7 GiB/s
    6 core: 70.1 GiB/s
    9 core: 82.7 GiB/s

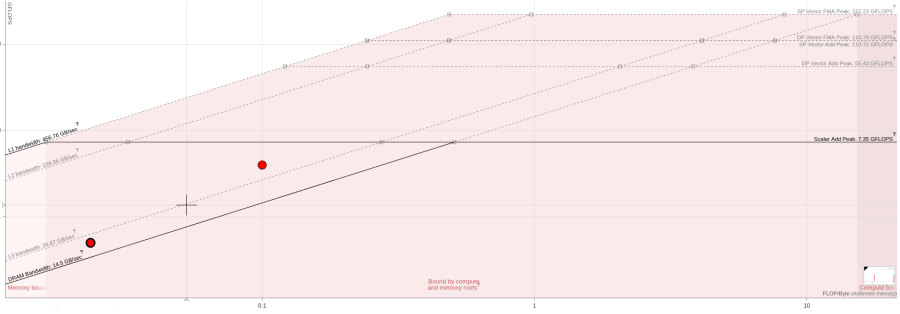


- Peak performance measurement
  - Using a compute bound kernel
  - Using dgemm:
    1 core: 98.0 GFLOP/s
    14 cores: 965.0 GFLOP/s
- Bandwidth measurement
  - Using a memory bound kernel
  - Using stream (triad):
    1 core: 12.7 GiB/s
    6 core: 70.1 GiB/s
    9 core: 82.7 GiB/s



Event	Latency	Scaled	Capacity
1 CPU cycle	0.1 ns	1 s	–
L1 cache access	1 ns	10 s	kB
L2 cache access	1 ns	10 s	MB
L3 cache access	10 ns	1 min	MB
RAM access	100 ns	10 min	GB
Solid-state disk access	100 μs	10 days	TB
Hard-disk drive access	1–10 ms	1–12 months	TB

Roofline model
Intel Amplifier



Profiling

A precious ally for optimization

- Where is my application spending most of the time?
  - (bad) measure time "by hand" using timings and prints
  - (good) use a tool made for this, e.g., Intel VTune Profiler, Score-P, gprof

- In addition to timings, profilers give you a lot more information on
  - Memory usage
  - Hardware counters
  - CPU activity
  - MPI communications
  - etc.

- Profile a code without bugs!
- Choose the right problem size (representative of your simulations)
- Focus first on the functions taking most of the time
- If the profile is not explicit, try refactoring into smaller functions
  - Some profilers, e.g., ScoreP, let you define custom regions

Pareto principle

The 80/20 rule

- General principle that states that 80% of the effect comes from 20% of causes
- Applies in many domains and, in particular, in optimization
- 80% of the time is spent in 20% of your code
- Concentrate on that 20% and don't optimize arbitrarily

- For the purpose of this exercise, we will use MiniFE
  - 3D implicit finite-elements on an unstructured mesh
  - C++ mini application
  - https://github.com/Mantevo/miniFE
  - You don't need to understand what the code does!
- We will use Intel VTune Profiler, part of the oneAPI Base toolkit (free)
- Connect to helvetios with X display options (macOS users may need to install XQuartz, and Windows users may need to install Xming)

```
$> ssh -X user@helvetios.hpc.epfl.ch
```

- Download miniFE
- Compile the basic version found in `ref/src`
- Profile the code using the hotspot analysis
- Open Intel VTune Profiler and select your timings
- Play around and find the 5 most time-consuming functions

- Download miniFE

```
$> git clone https://github.com/Mantevo/miniFE.git
$> cd miniFE
```

- Compile the basic version found in **ref/src**
  - You will need to load a compiler and an MPI library

```
$> module load intel intel-openapi-mpi intel-openapi-vtune
```

  - Change the `Makefile` to set `CXX=mpiicpc` and `CC=mpiicc` and compile

```
$> make
```

  - Make sure to compile your code with `-g -O3`

- Profile the code using

```
$> srun -n 1 vtune -collect hotspots -r prof_results --
↪  ./miniFE.x -nx 128 -ny 128 -nz 128
```

- This will profile for the "hotspots" and store the timings in `prof_results`
- You can have more info on the types of analysis with

```
$> vtune -h collect
```

- Open an interactive session in helvetios

```
$> Sinteract
```

- Open Intel VTune and select your timings

```
$> vtune-gui prof_results/prof_results.vtune
```

- 50.0% of the time spent in matrix/vector multiplications
- 12.5% of time spent imposing boundary conditions
- etc.
- Does the problem size influence the timings?

- This time, we profile a problem of size $(16, 16, 16)$
- 13.6% of the time is spent opening libraries
- 13.6% of the time is spent initializing MPI
- etc.
- Depending on the problem size, different parts of the code will dominate

Optimization

- We now have a pretty good idea of which part of the code to optimize
- Different options are possible (by order of complexity)
  1. Compiler and linker flags
  2. Optimized external libraries
  3. Handmade optimization (loop reordering, better data access, etc.)
  4. Algorithmic changes

Optimization

- We now have a pretty good idea of which part of the code to optimize
- Different options are possible (by order of complexity)
  1. Compiler and linker flags
  2. Optimized external libraries
  3. Handmade optimization (loop reordering, better data access, etc.)
  4. Algorithmic changes

- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc

- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc
- Common options are:
  - `-O0`, `-O1`, `-O2`, `-O3`: from almost no optimizations to most optimizations

- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc
- Common options are:
  - `-O0`, `-O1`, `-O2`, `-O3`: from almost no optimizations to most optimizations
  - `-Ofast`: activate more aggressive options, e.g., `-ffast-math` (but can produce wrong results in some particular cases)

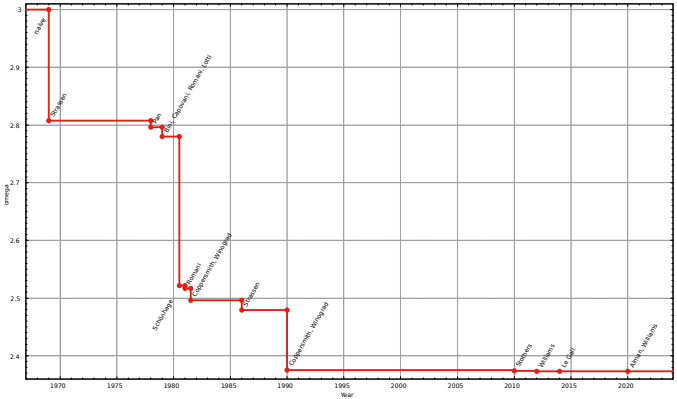
- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc
- Common options are:
  - `-O0`, `-O1`, `-O2`, `-O3`: from almost no optimizations to most optimizations
  - `-Ofast`: activate more aggressive options, e.g., `-ffast-math` (but can produce wrong results in some particular cases)
- Test your program with different options (`-O3` does not necessarily leads to faster programs)
- Note that the more optimization the longer the compilation time

- Do not re-invent the wheel!
- A lot of optimized libraries exist with different purposes (solvers, data structures, I/O, etc.). A few examples:
  - Solvers: PETSc, MUMPS, LAPACK, scaLAPACK, PARDISO, etc.
  - I/O: HDF5, ADIOS, etc.
  - Math libraries: FFTW, BLAS, etc.

- Sometimes, we cannot rely on compiler options or libraries and we must optimize "by hand"
- Usually, the goal is to rewrite the code in such a way that the compiler can optimize it
- Start by having a correct program before trying to optimize
- "Premature optimization is the root of all evil", D. Knuth

- Example of matrix/matrix multiplication. Graph shows complexity ($\mathcal{O}(n^{\omega})$) for different algorithms

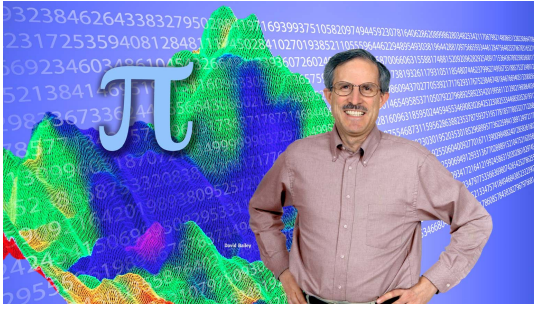
- Example of matrix/matrix multiplication. Graph shows complexity ($\mathcal{O}(n^\omega)$) for different algorithms



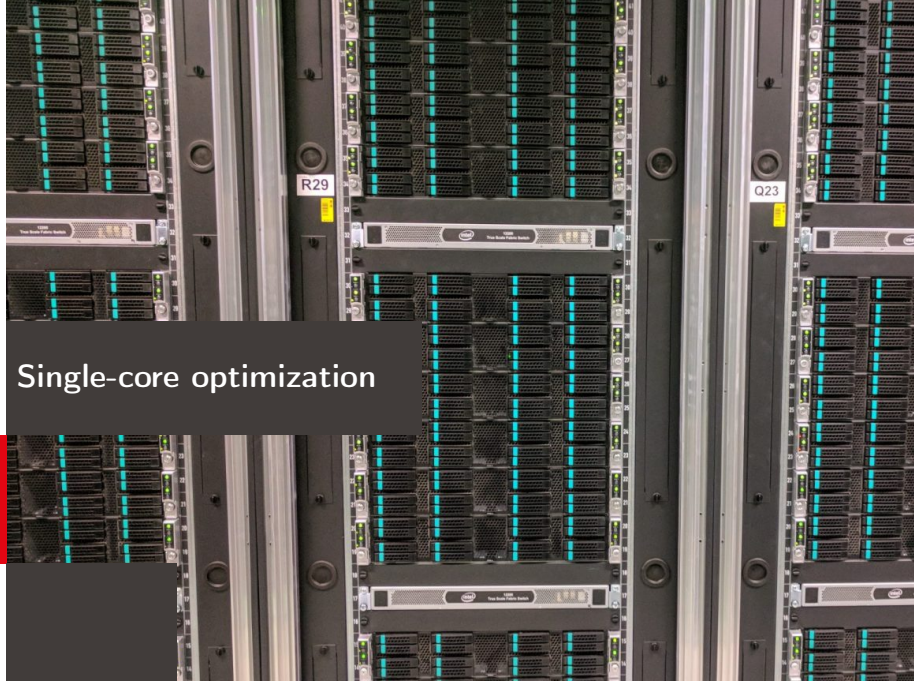
- Only when your code has *no bugs* and is *optimized*
- Are your ready to parallelize?
  1. Is it worth to parallelize my code? Does my algorithm scale?
  2. Performance prediction?
  3. Profiling?
  4. Bottlenecks?
  5. Which parallel paradigm should I use? What is the target architecture (SMP, cluster, GPU, hybrid, etc)?

In 1991, David H. Bailey published a famous paper: Twelve ways to fool the masses when giving performance results on parallel computers

6: Compare your results against scalar, unoptimized code on Crays.



Single-core optimization



- Better grasp how programming can influence performance
- We first review some basic optimization principles to keep in mind
- Deeper understanding of the working principles of the CPU
  - How data transfers are handled
  - Concept of vectorization

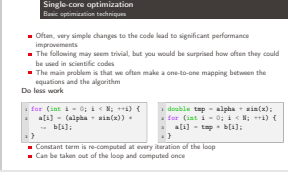
- Often, very simple changes to the code lead to significant performance improvements
- The following may seem trivial, but you would be surprised how often they could be used in scientific codes
- The main problem is that we often make a one-to-one mapping between the equations and the algorithm

Do less work

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = (alpha + sin(x)) *
    ↪  b[i];
3 }
```

```
1 double tmp = alpha + sin(x);
2 for (int i = 0; i < N; ++i) {
3   a[i] = tmp * b[i];
4 }
```

- Constant term is re-computed at every iteration of the loop
- Can be taken out of the loop and computed once



- In very simple cases like here, the compiler is smart enough to do it for you
- The main point is that the compiler will do most of the optimization job. Our goal is to write code that expresses our intention in a clear way so that the compiler can optimize it.

Avoid branches

```
1  for (i = 0; i < N; ++i) {
2    for (j = 0; j < N; ++j) {
3      if (j >= i) {
4        sign = 1.0;
5      } else {
6        sign = -1.0;
7      }
8      b[j] = sign * a[i][j];
9    }
10 }
```

```
1  for (i = 0; i < N; ++i) {
2    for (j = i; j < N; ++j) {
3      b[j] = a[i][j];
4    }
5    for (j = 0; j < i; ++j) {
6      b[j] =  -a[i][j];
7    }
8  }
```

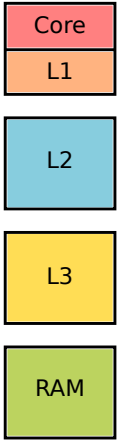
- Avoid conditional branches in loops
- They can often be written differently or taken out of the loop

- To better understand the concepts behind caching, let's take the example of a librarian
- The first customer enters and asks for a book. The librarian goes into the huge storeroom and returns with the book when he finds it
- After some time, the client returns the book and the librarian puts it back into the storeroom
- A second customer enters and asks for the same book...
- This workflow can take a lot of time depending on how much customers want to read the same book

- Our librarian is a bit lazy, but clever. Since a lot of customers ask for the same book, he decides to put a small shelf behind his desk to temporarily store the books he retrieves.
- This way he can quickly grab the book instead of going to the storeroom.
- When a customer asks for a book, he will first look on his shelf. If he finds the book, it's a *cache hit* and he returns it to the customer. If not, it's a *cache miss* and he must go back in the storeroom.
- This is a very clever system, especially if there is *temporal locality*, i.e., if the customers often ask for the same books.
- Can he do better?

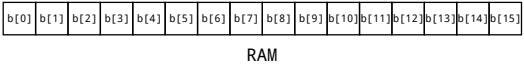
- Oftentimes, our librarian see that people taking one book will go back and ask for the sequels of the book
- He decides to change a bit his workflow. Now, when he goes into the storeroom to retrieve a book, he comes back with a few of them, all on the same shelf
- This way, when the customer brings back a book and asks for the sequel, it is already present on the librarian shelf
- This workflow works well when there is *spatial locality*, i.e., when you ask for a book there is a significant chance that you will read the sequel

- Now, what is the link between our librarian and the CPU? They work in a similar fashion!
- When a load instruction is issued the L1 cache logic checks if data is already present. If yes, this is a *cache hit* and data can be retrieved very quickly. If no, this is a *cache miss* and the next memory levels are checked.
- If the data is nowhere to be found, then it is loaded from the main memory
- As for our librarian, not only the required data is loaded for each cache miss, but a whole *cache line*



- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

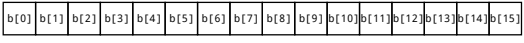


- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

Load b[0]:
Cache miss
Fetch cache line from RAM

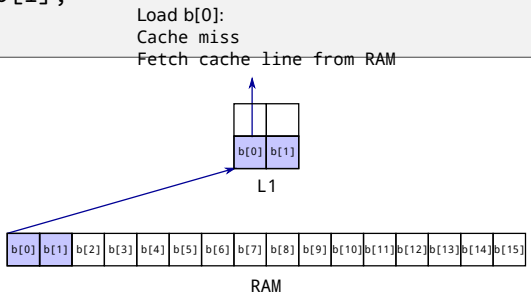
L1



RAM

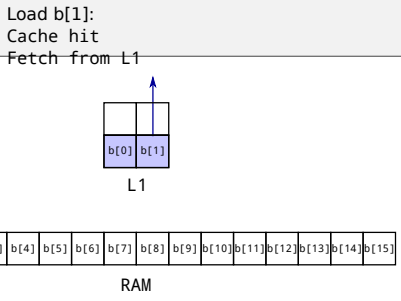
- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```



- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

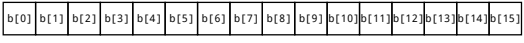


- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

Load b[2]:
Cache miss
Fetch cache line from RAM

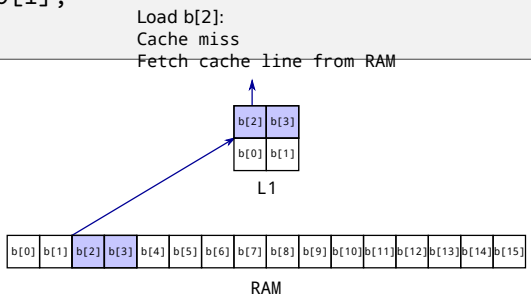
L1



RAM

- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

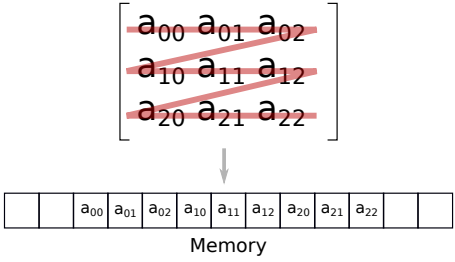


- How do we store ND arrays into memory?
- Memory is a linear storage. Arrays are stored contiguously, one element after the other.
- We have to choose a convention. Row major (C/C++) or column major (Fortran).
- Row major means that elements are stored contiguously according to the last index of the array. In column-major order, they are stored according to the first index.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

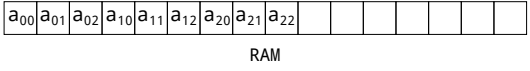
Memory

- How do we store ND arrays into memory?
- Memory is a linear storage. Arrays are stored contiguously, one element after the other.
- We have to choose a convention. Row major (C/C++) or column major (Fortran).
- Row major means that elements are stored contiguously according to the last index of the array. In column-major order, they are stored according to the first index.



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

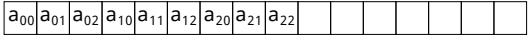


- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

Load a[0][0]:
Cache miss
Fetch cache line from RAM

L1

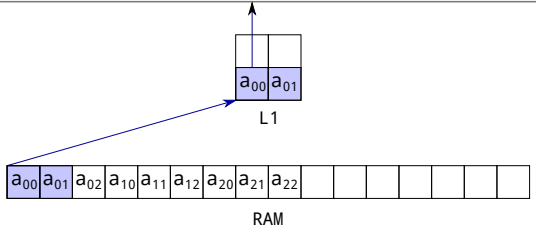


RAM

- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

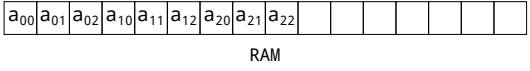
Load a[0][0]:
Cache miss
Fetch cache line from RAM



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

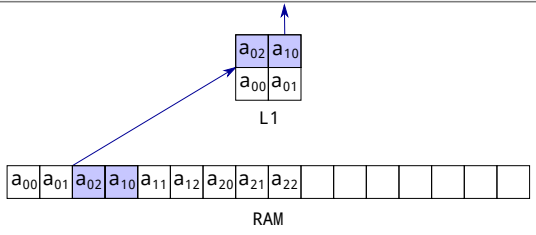
Load a[1][0]:
Cache miss
Fetch cache line from RAM



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

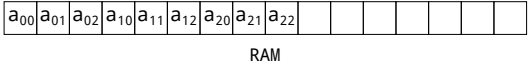
Load a[1][0]:
Cache miss
Fetch cache line from RAM



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

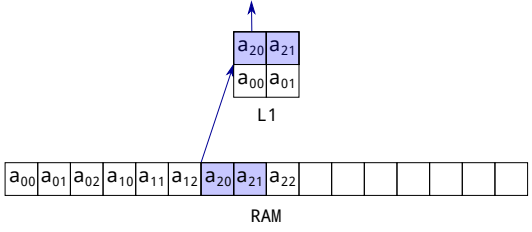
Load a[2][0]:
Cache miss
Fetch cache line from RAM



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

Load a[2][0]:
Cache miss
Fetch cache line from RAM

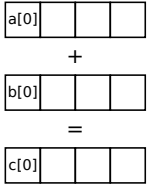


- Caches are small, but very fast memories
- Their purpose is to alleviate long latency and limited bandwidth of the RAM
- Data is fetched by group, called cache line, and stored into the different levels of cache
- In order to fully exploit caches, data in caches must be re-used as much as possible (temporal locality)

- Avoid random memory accesses that case many cache misses and prefer contiguous access (spatial locality)
- Be careful of the data types you use and how they are mapped onto memory

- Modern CPUs can apply the same operation to multiple data
- Special registers `xmm`, `ymm` and `zmm` holding 2, 4 or 8 doubles

```
for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i]
}
```



- Modern CPUs can apply the same operation to multiple data
- Special registers `xmm`, `ymm` and `zmm` holding 2, 4 or 8 doubles

```
for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i]
}
```