Intro to Modern Programmable Architectures

Focus on Single-core CPU

Sorbonne Université – Master SESI – MU5IN60 – Parallel Programming

Adrien CASSAGNE

September 18, 2023

Table of Contents

- 6 ECTS, 11 sessions of 4 hours:
  - — 6 sessions of **classroom lectures** ($6 \times 2h$)
  - — 6 sessions of **hands-on works** ($6 \times 2h$)
  - — 4 sessions of a **project** ($4 \times 4h$)
  - — 1 session for the **presentations** ($1 \times 4h$)

- Main objectives:
  - — Understand **modern general purpose processor architectures** (CPU & GPU)
  - — **Program efficiently** the CPUs and GPUs
  - — Learn the theory to upper-bound the **expected performance benefit**

- Take away:
  1. CPU programming
     - ○ Scalar optimizations
     - ○ SIMD instructions
     - ○ Multi-threading
  2. GPU programming
     - ○ Concepts
     - ○ OpenCL language
  3. Performance analysis
     - ○ Speedup
     - ○ Efficiency
     - ○ Roofline model

- Sessions in **English**

- Strong basis in **C programming**
  — Pointers
  — Structures
  — Compilation and Makefile
  — ...

- Few basis in **C++ programming**
  — Standard Template Library
  — Ex.: std::vector<T>
  — Templating over the data types

- Some knowledge about **computers architecture**
  — Central Unit Processor (CPU)
  — RAM, Caches, Bus
  — Main principles of assembly prog.

- Some knowledge about **Linux OS**
  — Command line
  — Memory pagination and space
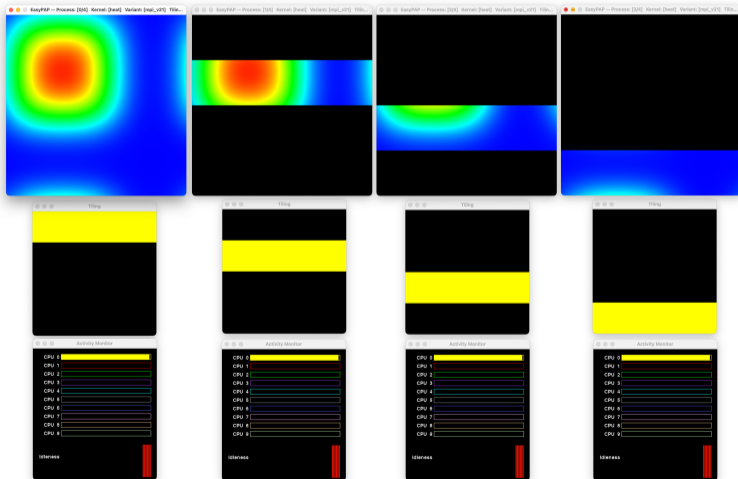 — Unix processes and threads

- **NVIDIA Jetson TX2** development kit
  — CPU: 6 cores
  — GPU: 256 cores
  — RAM: 8 GB shared between CPU & GPU
  — **Embedded in cars, satellites, video game consoles (Switch), medical equipment, ...**
- **15 available boards** (max. 2 students per board)
- Hands-on and project sessions on **the same target!**
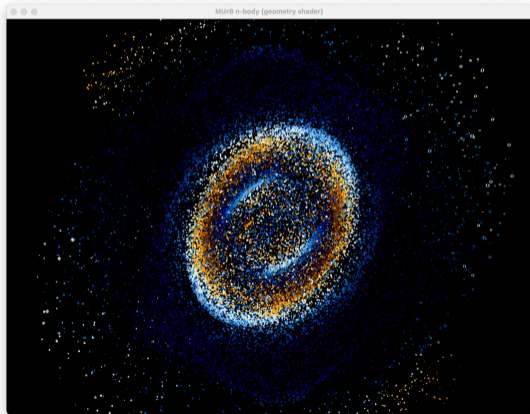
Application Framework

- Hands-on sessions with **EasyPAP framework**
- **C** and **C++** languages
- **Visualization** of the results
- **Monitoring** of the performance
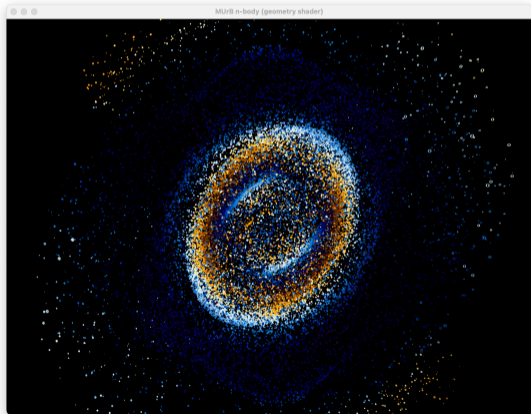- **Focus on what really matters**

- **Surprise!**
- **Choose ProgPar to find out more...**

- **Surprise!**
- **Choose ProgPar to find out more...**
- Oh but you already chose ProgPar, be patient you will see later ;-)
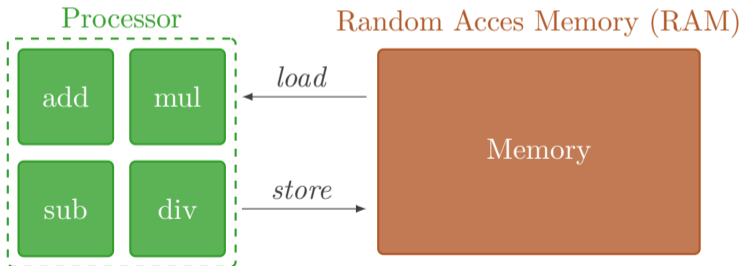
Table of Contents

- Compute units and memory
  - Memory is used to load and store data
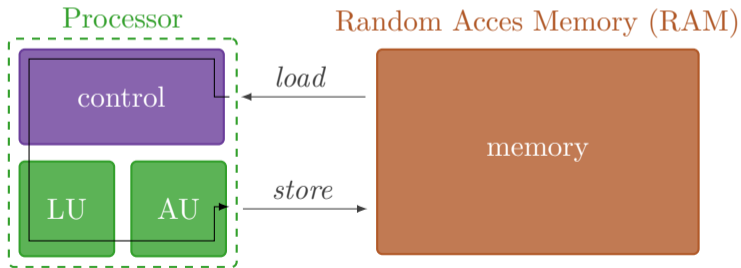 - Compute units are used to transform data

Simplified CPU architecture – Part 2

2 Programmable Architectures

- More precisely, in a CPU there are:
  - — Control units (`if`, `goto`, instructions placement, ...)
  - — Logic units, LU (==, ! =, >, <, ...)
  - — Arithmetic units, AU (+, ∗, −, /, ...)

CPU Architecture: Cycle, Freq. and Throughput

2 Programmable Architectures

- A tick of the clock = a cycle
- In each cycle, the CPU performs an elementary task
- Frequency = number of cycles per second (in Hertz)

CPU Architecture: Cycle, Freq. and Throughput

- A tick of the clock = a cycle

- In each cycle, the CPU performs an elementary task

- Frequency = number of cycles per second (in Hertz)

- Modern CPU/RAM frequency: between 1 GHz and 4 GHz (1 GHz = 10^9 Hz)

- Memory throughput (RAM) $\approx$ 50 GB/s (DDR5)

- CPU capable of consuming/producing data at $\approx$ 5 TB/s (i9-9900K)
  - **The CPU is much faster ($\times 100$) !**
  - How can we solve this problem?

Observation: most applications often reuse the same data

**Observation: most applications often reuse the same data**

- Faster memory between CPU and RAM = **Cache memory**
    — Faster memory is more expensive and takes up physical space
    — This memory (= cache) is much smaller than RAM
    — 3 levels of cache (in the processor):
        ○ L1, the fastest and smallest (32 KB), access time ≈ 2 cycles
        ○ L2, slower than L1 but larger (1 MB), access time ≈ 10 cycles
        ○ L3, slower than L2 but larger (4 MB), access time ≈ 30 cycles
    — RAM access latency ≈ 100 cycles
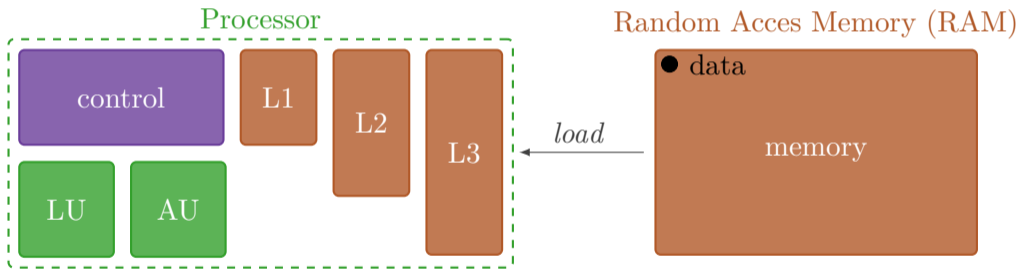
- Scenario: **first** *load* of a data from memory

Loading data from RAM: ≈ **100 cycles**

- Scenario: **first** *load* of a data from memory
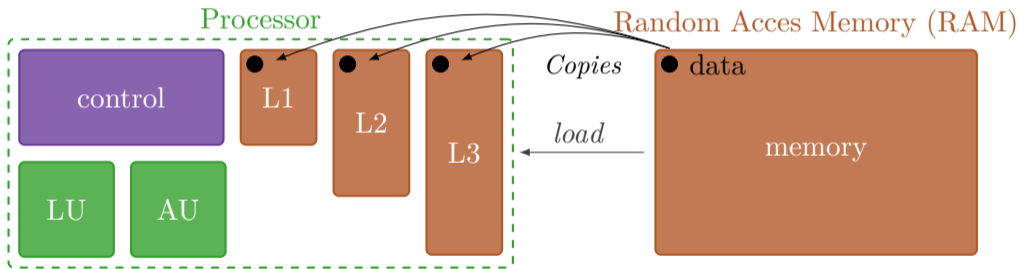
Loading data from RAM: $\approx$ **100 cycles**

- Scenario: *load* the same data (= **second** load) from memory

Loading data from L1 cache: **1-2 cycles**

- Scenario: *load* the same data (= **second** load) from memory

Processor

Random Acces Memory (RAM)

control

L1

L2

L3

● data

load

memory

LU   AU

Loading data from L1 cache: **1-2 cycles**

- This is the principle of **temporal locality**

- Scenario: *store* data to memory

- Two possible modes:

  — **Write-through**
    - "Possibly simpler" to implement in hardware
    - Write to both RAM and cache
    - Choice made in some embedded architectures

  — **Write-back**
    - "More complicated" to implement in hardware (cache coherence protocol)
    - Write to cache and write to RAM only if data is invalidated from smaller/faster cache
    - Writing to RAM less often = consuming less energy
    - The choice made by the majority of architectures (computer, HPC, embedded)

- One cache line = a certain amount of data (e.g. 128–512 bits)
- The smallest packet of data moving between RAM and CPU is a cache line
- **It's more interesting to access data contiguously!**
- Otherwise, part of the data coming from RAM will be unused (= loss of bandwidth).

CPU Architecture: Spatial Locality
2 Programmable Architectures

- One cache line = a certain amount of data (e.g. 128–512 bits)
- The smallest packet of data moving between RAM and CPU is a cache line
- **It's more interesting to access data contiguously!**
- Otherwise, part of the data coming from RAM will be unused (= loss of bandwidth).
- Using data in the same cache line is called **spatial locality**!

- Scalar instruction: produces data during 1 cycle

$$r_a \quad + \quad r_b \quad = \quad r_c$$

- SIMD instruction: produces n data during 1 cycle

$$
\begin{array}{ccccc}
\boldsymbol{r_a} & \text{SIMD} & \boldsymbol{r_b} & & \boldsymbol{r_c} \\
r_a^0 & + & r_b^0 & = & r_c^0 \\
r_a^1 & + & r_b^1 & = & r_c^1 \\
r_a^2 & + & r_b^2 & = & r_c^2 \\
r_a^2 & + & r_b^3 & = & r_c^3
\end{array}
$$
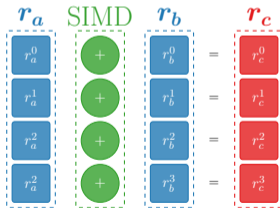
- SIMD instructions operate on so-called "vector registers".

- Modern CPUs are now all multi-core
- Generally, L1 and L2 caches are dedicated to a single core.
- Cores share the *Last Level Cache* (LLC) or L3

Multi-core processor

Core 0 — control, L1, L2, LU, AU

Core 1 — control, L1, L2, LU, AU

L3 Cache or *Last Level Cache* (LLC)

Core 2 — control, L1, L2, LU, AU

Core 3 — control, L1, L2, LU, AU
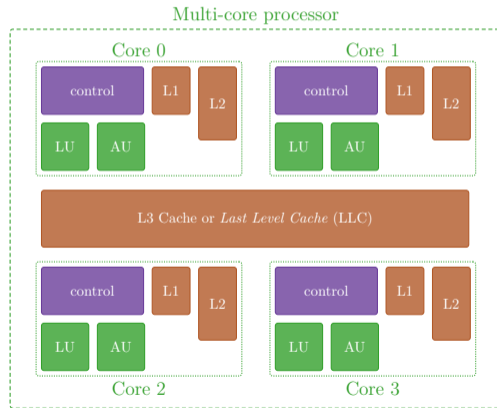
- Hardware physically separated from the CPU (or not on today's SoCs...).
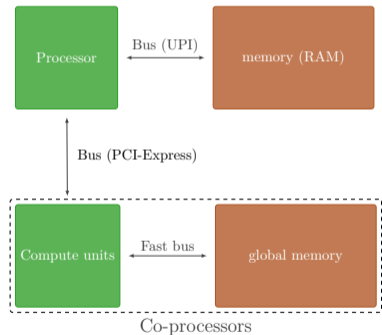- Often connected to the CPU via the PCI-Express bus
- Has its own RAM (global memory) (but not always...)
- Examples of co-processors
  — Graphics Processing Unit (GPU)
  — Field-Programmable Gate Array (FPGA)
  — Many Integrated Cores (MIC)

GPU Architecture
2 Programmable Architectures

- Originally designed for image processing
- Massively parallel architecture
- Fewer control units than CPUs, but more compute units
- Global memory faster than CPU RAM (≈ 500 GB/s)
- Performance/power consumption ratio generally better than CPUs
- Often suitable for scientific computing
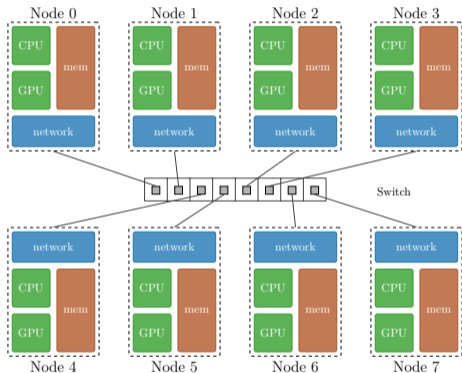- Hardware acceleration units for AI (tensor cores)

- A very simplistic supercomputer
- Interconnecting computers with a star network (switch)
- Theoretical maximum performance is 8 times that of a node

Table of Contents

- Generally, an instruction can be divided into several sub-steps:
  1. *Fetch* (IF) : the instruction is copied from memory
  2. *Decode* (ID) : the instruction is interpreted by the CPU
  3. *Execute* (EX) : the instruction is executed

- On a processor WITHOUT pipeline, this corresponds to:
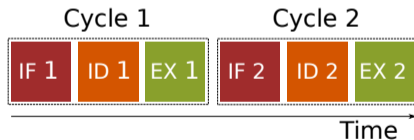
- Cycle 1 and cycle 2 take 6 units of time to complete

- From what we've just seen, it's possible to divide an instruction into 3 sub-instructions (μop)
- We assume that each sub-instruction takes the same amount of time
- On a pipelined processor, this corresponds to:

| IF 1 | ID 1 | EX 1 | Cycle 1 |

| IF 2 | ID 2 | EX 2 | Cycle 2 |

| IF 3 | ID 3 | EX 3 | Cycle 3 |
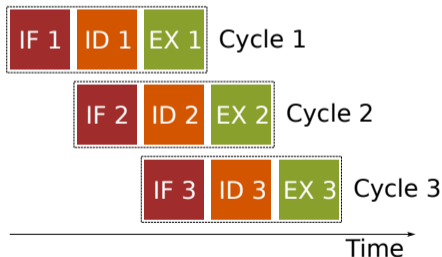
Time →

- Cycle 1, cycle 2 and cycle 3 take 5 units of time to complete
- 3-stage pipeline (IF, ID and EX can run in parallel)

- It takes some time before the pipeline is optimal
  — This is called pipeline latency (2 cycles here)

- If latency is not taken into account, a 3-stage pipeline is 3 times faster

- Yellow: the time point at which the pipeline is optimal

- Today's processors have pipelines of 10 to 20 stages, but the principle remains the same

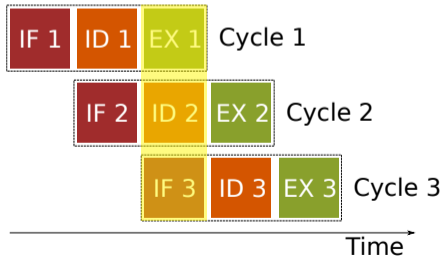- There is a mechanism so that the preceding instruction can directly pass its result to the following instruction (*register latch + forwarding*)

- Pipeline seems effective, but what happens if you have conditions (e.g. if)?
  — In this case, it's problematic: the processor can't know what the next instruction will be, which is bad to pipeline efficiency (it creates "bubbles").
  — There are branch prediction mechanisms to limit this effect...

- The ability of a processor to execute several instructions in parallel
  - *Instruction Level Parallelism* (ILP)
  - Processors are generally 3- to 10-wide super-scalar

- This means that a CPU is capable of executing more than one instruction per cycle (3 to 10 instructions)
  - The example on the right shows a processor with an ILP of 2 and a 3-stage pipeline
  - For instance, a processor can compute 1 multiplication and load data from memory in 1 cycle

IF 1 ID 1 EX 1 Cycle 1

IF 2 ID 2 EX 2 Cycle 2

IF 3 ID 3 EX 3 Cycle 3

IF 4 ID 4 EX 4 Cycle 1

IF 5 ID 5 EX 5 Cycle 2

IF 6 ID 6 EX 6 Cycle 3

Time

Out-of-Order (OoO) Execution

- Most processors are now capable of executing instructions out of order (different order than the one given by the assembler code)
  - This maximizes the use of CPU units during a cycle (e.g. super-scalar processor).
  - It is difficult to predict the order in which the processor will execute instructions

- Here's a code example to illustrate:

```
1 int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h = 8;
2 c = a + b; // 1st instruction to be executed, it depends on nothing
3 e = c * d; // 3rd instruction to be executed, it depends on 'c'
4 h = f * g; // 2nd instruction to be executed, it depends on nothing
```

Out-of-Order (OoO) Execution

3 Single-core CPU Architecture

- Most processors are now capable of executing instructions out of order (different order than the one given by the assembler code)
  - This maximizes the use of CPU units during a cycle (e.g. super-scalar processor).
  - It is difficult to predict the order in which the processor will execute instructions

- Here's a code example to illustrate:

```
1 int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h = 8;
2 c = a + b; // 1st instruction to be executed, it depends on nothing
3 e = c * d; // 3rd instruction to be executed, it depends on 'c'
4 h = f * g; // 2nd instruction to be executed, it depends on nothing
```
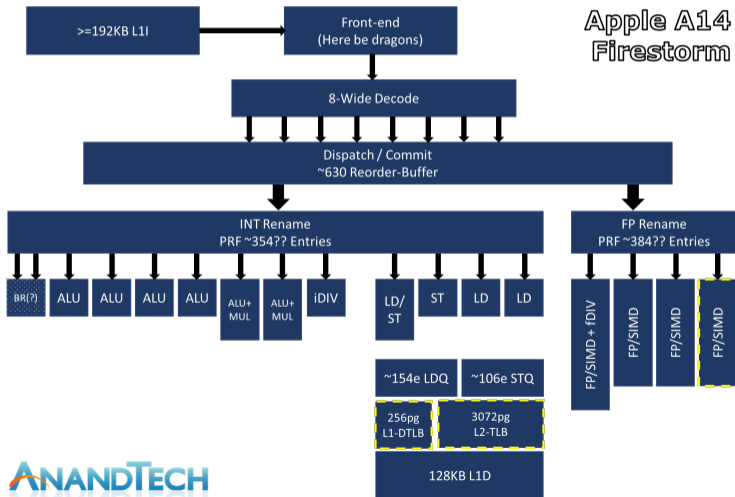
- Most processors are now capable of executing instructions out of order
  (different order than the one given by the assembler code)
  — This maximizes the use of CPU units during a cycle (e.g. super-scalar processor).
  — It is difficult to predict the order in which the processor will execute instructions

- Here's a code example to illustrate:

```
1 int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h = 8;
2 c = a + b; // 1st instruction to be executed, it depends on nothing
3 e = c * d; // 3rd instruction to be executed, it depends on 'c'
4 h = f * g; // 2nd instruction to be executed, it depends on nothing
```

Out of Order Engine

Track μop dependencies and dispatch ready μops to execution units

Wider
5 → 6 wide allocation
10 → 12 execution ports

Deeper
512-entry Reorder-Buffer and larger
Scheduler sizes

Smarter
More instructions "executed" at
rename / allocation stage

- **Nvidia Jetson TX2** development kit
  - — 2× very powerful Nvidia "Denver 2" cores @ 2.04 GHz
    - ○ Specific for performance applications
  - — 4× powerful ARM "Cortex A57" (Big) cores @ 2.04 GHz
    - ○ Generally used in smartphones
  - — Weird heterogeneous design
    - ○ Classic ARM architectures combine A57 (Big) with A53 (LITTLE)
  - — No energy efficient cores (LITTLE)

- 18-stage pipeline
- 3-way decoder
- 8-wide super-scalar (μops)
- Out-of-order execution
- Source: Anandtech
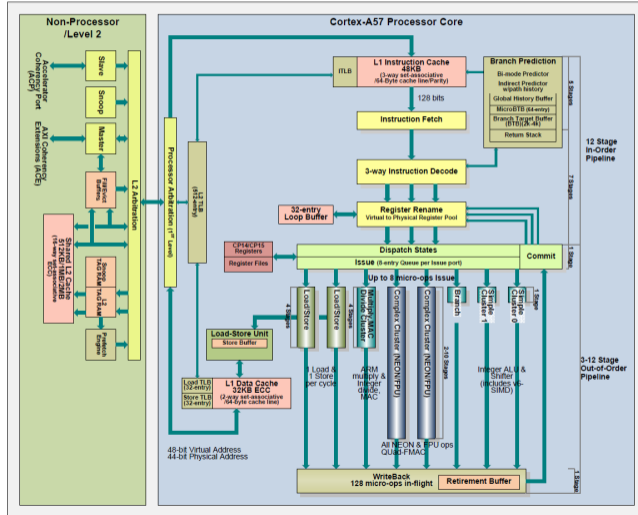
Nvidia Denver 2 (2016)

- 15-stage pipeline
- 2-way decoder
- 7-wide super-scalar (μops)
- In-order execution
  — ARM code is translated by a hardware translator
  — Can be considered as out-of-order...
- Sources:
  — wccftech
  — WikiChip
  — Wikipedia
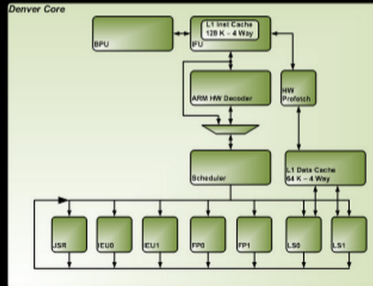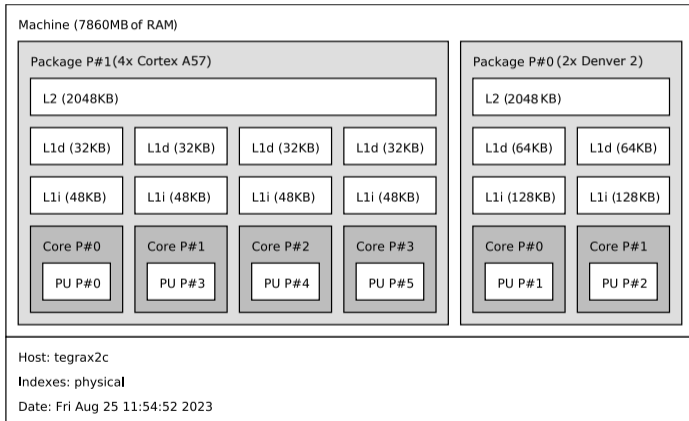
Topology from `hwloc-ls`

- Compilers come with a number of options that allow you to optimize your program automatically, thus reducing execution time

- In this course, we'll be using the GNU C/C++ compiler (gcc, g++) or Clang (clang, clang++), but similar options are available in other compilers

- It's important to understand the optimizations that can and cannot be made by the compiler!
  — This maximizes the readability of the source code while maintaining efficiency
  — This allows the compiler to apply some "dirty" optimizations on our behalf when generating the binary code

Compiler: Optimization Options
4 Single-core CPU Optimizations

- The most famous are (`-O[level]`) :
  - `-O0` no optimization
  - `-O1` enables a series of optimizations to reduce binary size and execution time, while keeping compilation time relatively low
  - `-O2` enables all possible optimizations (except those requiring a compromise between efficiency and binary size), this option requires a longer compilation time than `-O1`
  - `-O3` optimizes even further, enables the following options: `-finline-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-after-reload`, `-ftree-loop-vectorize`, `-ftree-loop-distribute-patterns`, `-ftree-slp-vectorize`, `-fvect-cost-model`, `-ftree-partial-pre` and `-fipa-cp-clone`

- Source: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- `-finline-functions`: enables automatic inlining, the compiler can choose whether or not to inline (this option is not a guarantee)

- `-ftree-vectorize`: enables automatic code vectorization

- `-ffast-math`: does not take IEEE 754 specifications into account in float calculations (risk of loss of precision = risk of bugs)

- `-funroll-loops`: unrolls loops whose bounds are known at compile time. This option makes the binary code larger, and does not necessarily improve execution time

- `-march=native`: enables instructions specific to the micro-architecture on which the compiler is running, often necessary for code vectorization

Instruction Latency and Throughput

- Here are the costs of the main arithmetic instructions on the *Intel Skylake* micro-architecture
  - add: latency of 4 cycles, throughput of 0.5 cycles per instruction (CPI)
  - sub: latency of 4 cycles, throughput of 0.5 cycles per instruction (CPI)
  - mul: latency of 4 cycles, throughput of 0.5 cycles per instruction (CPI)
  - div: latency of 14 cycles, throughput of 4 cycles per instruction (CPI)
  - Source: Intel Intrinsics Guide

- For add, sub and mul, the CPU can achieve 2 instructions per cycle!

- As you can see, division is much less efficient than multiplication (throughput 8 times lower)
  - It is therefore interesting to compute the inverse and then multiply by the inverse
  - Be careful, however, as this operation leads to a loss of precision in the computations

```
1 int div1(const float *A, float *B, const int n) {
2   for (int i = 0; i < n; i++)
3     B[i] = A[i] / 3.f;
4 }
```

- Number of theoretical cycles: $n \times 4$

```
1 int div2(const float *A, float *B, const int n) {
2   float inv3 = 1.f / 3.f;
3   for (int i = 0; i < n; i++)
4     B[i] = A[i] * inv3;
5 }
```

- Number of theoretical cycles: $4 + n \times 0.5$

- Compiler: `Clang 10.0.0` – Options: `-O1 -march=armv8-a+nosimd`

```
 1 div1:
 2         cmp     w2, #1
 3         b.lt    .LBB0_3
 4         mov     w8, w2
 5         fmov    s0, #3.00000000
 6 .LBB0_2: // =>This Inner Loop Header: D=1
 7         ldr     s1, [x0], #4
 8         subs    x8, x8, #1
 9         fdiv    s1, s1, s0
10         str     s1, [x1], #4
11         b.ne    .LBB0_2
12 .LBB0_3:
13         ret
```

```
 1 div2:
 2         cmp     w2, #1
 3         b.lt    .LBB0_3
 4         mov     w9, #43691
 5         mov     w8, w2
 6         movk    w9, #16042, lsl #16
 7 .LBB0_2: // =>This Inner Loop Header: D=1
 8         ldr     s0, [x0], #4
 9         fmov    s1, w9
10         subs    x8, x8, #1
11         fmul    s0, s0, s1
12         str     s0, [x1], #4
13         b.ne    .LBB0_2
14 .LBB0_3:
15         ret
```

Special Functions

- Here are the costs of some common mathematical functions (*Intel Skylake*):
  - sqrt: latency of 12 cycles, throughput of 3 CPI
  - rsqrt: latency of 4 cycles, throughput of 1 CPI
  - pow, cos, sin, tan: very expensive, depends on software implementation, no dedicated hardware unit
  - Source: Intel Intrinsics Guide

- rsqrt has a throughput of one instruction per cycle!
  - Surprisingly, this function is implemented in hardware
  - Accuracy is generally lower than for other instructions
  - Widely used for 2D/3D distance calculations

- pow, cos, sin and tan are very expensive, we must try to limit their use in our codes
  - Approximate functions are often available at lower cost

- A function call comes with an additional cost (passing parameters through registers and/or the stack, jump instructions)
- Does this mean we shouldn't make function calls?
  — Sometimes it's better to avoid
  — It depends on where the function call is located

```
1 void stencil_1d_core(const float *A, float *B, const int i) {
2   B[i] = A[i - 1] + A[i ] + A[i + 1];
3 }
4
5 void stencil_1d(const float *A, float *B, const int size) {
6   for (int i = 1; i < size -1; i++)
7     stencil_1d_core(A, B, i); // ici on appelle la fonction très souvent
8 }
```

- Inlining a function means replacing the function call with the function code itself
  — This eliminates the extra cost of calling the function

- You can do it manually, but it's not a good idea...

- It's a better idea to have the compiler do it!
  — Use of the keyword `inline` in the C/C++ language
  — Compiler optimization option (`-finline-function`)
  — In the code before the function declaration: `__attribute__((inline))`

```
1   stencil_1d_core:
2           sbfiz   x8, x2, #2, #32
3           add     x9, x0, x8
4           ldp     s0, s1, [x9, #-4]
5           ldr     s2, [x9, #4]
6           fadd    s0, s0, s1
7           fadd    s0, s0, s2
8           str     s0, [x1, x8]
9           ret
10  stencil_1d:
11          stp     x29, x30, [sp, #-48]!
12          stp     x22, x21, [sp, #16]
13          stp     x20, x19, [sp, #32]
14          mov     x29, sp
15          cmp     w2, #3
16          b.lt    .LBB1_3
17          mov     w19, w2
18          mov     x20, x1
19          mov     x21, x0
20          mov     w22, #2
21  .LBB1_2: // =>This Inner Loop Header: D=1
22          sub     w2, w22, #1
23          mov     x0, x21
24          mov     x1, x20
25          bl      stencil_1d_core
26          add     w22, w22, #1
27          cmp     w19, w22
28          b.ne    .LBB1_2
29  .LBB1_3:
30          ldp     x20, x19, [sp, #32]
31          ldp     x22, x21, [sp, #16]
32          ldp     x29, x30, [sp], #48
33          ret
```

```
1   stencil_1d_core:
2           sbfiz   x8, x2, #2, #32
3           add     x9, x0, x8
4           ldp     s0, s1, [x9, #-4]
5           ldr     s2, [x9, #4]
6           fadd    s0, s0, s1
7           fadd    s0, s0, s2
8           str     s0, [x1, x8]
9           ret
10  stencil_1d:
11          cmp     w2, #3
12          b.lt    .LBB1_3
13          sub     w10, w2, #1
14          add     x8, x0, #8
15          add     x9, x1, #4
16          sub     x10, x10, #1
17  .LBB1_2: // =>This Inner Loop Header: D=1
18          ldp     s0, s1, [x8, #-8]
19          ldr     s2, [x8], #4
20          subs    x10, x10, #1
21          fadd    s0, s0, s1
22          fadd    s0, s0, s2
23          str     s0, [x9], #4
24          b.ne    .LBB1_2
25  .LBB1_3:
26          ret
```

- Compiler: Clang 10.0.0 – Options:
  -O2 -march=armv8-a+nosimd

Loop Unrolling

- Loop unwinding consists in increasing the loop pitch and adapting the loop body to the loop pitch

- Can sometimes be performed by the compiler (but not always...)

- Several benefits
  — Reduces time spent in loop control
  — Reduces the risk of branch prediction error
  — Increases optimization opportunities, potentially exposes more parallelism for ILP, masks instruction latency

- Some drawbacks
  — Reduces code readability and increases the risk of bugs (not good for maintainability)
  — An epilogue (code after the loop) is often required

Initial code without loop unrolling

```
 1 void basic_loop1(const float *A,
 2                  const float *B,
 3                  const float *C,
 4                  const int n)
 5 {
 6     for (int i = 0; i < n; i++) {
 7         D[i] = A[i] + B[i] + C[i];
 8     }
 9 }
```

Code with 2nd-order loop unrolling

```
 1 void basic_loop2(const float *A,
 2                  const float *B,
 3                  const float *C,
 4                  const int n)
 5 {
 6   for (i = 0; i < n; i += 2) {
 7     D[i + 0] = A[i + 0] + B[i + 0] + C[i + 0];
 8     D[i + 1] = A[i + 1] + B[i + 1] + C[i + 1];
 9   }
10 }
```

Initial code without loop unrolling

```
1 void basic_loop1(const float *A,
2                  const float *B,
3                  const float *C,
4                  const int n)
5 {
6     for (int i = 0; i < n; i++) {
7         D[i] = A[i] + B[i] + C[i];
8     }
9 }
```

Code with 2nd-order loop unrolling

```
1 void basic_loop2(const float *A,
2                  const float *B,
3                  const float *C,
4                  const int n)
5 {
6   for (i = 0; i < n; i += 2) {
7     D[i + 0] = A[i + 0] + B[i + 0] + C[i + 0];
8     D[i + 1] = A[i + 1] + B[i + 1] + C[i + 1];
9   }
10 }
```

- If we assume that n value is 3, then the code with unrolling is wrong!

Initial code without loop unrolling

```
1 void basic_loop1(const float *A,
2                  const float *B,
3                  const float *C,
4                  const int n)
5 {
6     for (int i = 0; i < n; i++) {
7         D[i] = A[i] + B[i] + C[i];
8     }
9 }
```

Code with 2nd-order loop unrolling

```
1 void basic_loop2(const float *A,
2                  const float *B,
3                  const float *C,
4                  const int n)
5 {
6   for (i = 0; i < (n * 2) / 2; i += 2) {
7     D[i + 0] = A[i + 0] + B[i + 0] + C[i + 0];
8     D[i + 1] = A[i + 1] + B[i + 1] + C[i + 1];
9   }
10  if (n % 2)
11    D[n - 1] = A[n - 1] + B[n - 1] + C[n - 1];
12 }
```

- If we assume that n value is 3, then the code with unrolling is wrong!
- We nee to add an epilogue L10-11

```
 1  basic_loop1:
 2          cmp     w4, #1
 3          b.lt    .LBB0_3
 4          mov     w8, w4
 5  .LBB0_2: // =>This Inner Loop Header: Depth=1
 6          ldr     s0, [x0], #4
 7          ldr     s1, [x1], #4
 8          ldr     s2, [x2], #4
 9          subs    x8, x8, #1
10          fadd    s0, s0, s1
11          fadd    s0, s0, s2
12          str     s0, [x3], #4
13          b.ne    .LBB0_2
14  .LBB0_3:
15          ret
```

```
 1  basic_loop2:
 2          subs    w8, w4, #1
 3          b.lt    .LBB0_3
 4          mov     x9, xzr
 5          mov     w10, w4
 6          add     x11, x2, #4
 7          add     x12, x1, #4
 8          add     x13, x0, #4
 9          add     x14, x3, #4
10  .LBB0_2: // =>This Inner Loop Header: Depth=1
11          ldur    s0, [x13, #-4]
12          ldur    s1, [x12, #-4]
13          ldur    s2, [x11, #-4]
14          add     x9, x9, #2
15          cmp     x9, x10
16          fadd    s0, s0, s1
17          fadd    s0, s0, s2
18          stur    s0, [x14, #-4]
19          ldr     s0, [x13], #8
20          ldr     s1, [x12], #8
21          fadd    s0, s0, s1
22          ldr     s1, [x11], #8
23          fadd    s0, s0, s1
24          str     s0, [x14], #8
25          b.lo    .LBB0_2
26  .LBB0_3:
27          tbz     w4, #0, .LBB0_5
28          sbfiz   x8, x8, #2, #32
29          ldr     s0, [x0, x8]
30          ldr     s1, [x1, x8]
31          ldr     s2, [x2, x8]
32          fadd    s0, s0, s1
33          fadd    s0, s0, s2
34          str     s0, [x3, x8]
35  .LBB0_5:
36          ret
```

- Compiler: Clang 10.0.0 – Options:

-O3 -march=armv8-a+nosimd -funroll-loops

- By default, the compiler does not unroll the code even with -funroll-loops

Loop Unrolling
4 Single-core CPU Optimizations

Initial code without loop unrolling

```c
1  void basic_loop1(const float *A,
2                   const float *B,
3                   const float *C,
4                   const int n)
5  {
6    #pragma unroll 2
7    for (int i = 0; i < n; i++) {
8      D[i] = A[i] + B[i] + C[i];
9    }
10 }
```

- Compiler: `Clang 10.0.0` – Options:

`-O3 -march=armv8-a+nosimd -funroll-loops`

```asm
1  basic_loop1:
2          cmp     w4, #1
3          b.lt    .LBB0_7
4          cmp     w4, #1
5          b.ne    .LBB0_3
6          mov     x8, xzr
7          b       .LBB0_6
8  .LBB0_3:
9          and     w9, w4, #0x1
10         mov     x8, xzr
11         add     x10, x0, #4
12         add     x11, x1, #4
13         add     x12, x2, #4
14         sub     w13, w4, w9
15         add     x14, x3, #4
16 .LBB0_4: // =>This Inner Loop Header: Depth=1
17         ldur    s0, [x10, #-4]
18         ldur    s1, [x11, #-4]
19         ldur    s2, [x12, #-4]
20         add     x8, x8, #2
21         cmp     w13, w8
22         fadd    s0, s0, s1
23         fadd    s0, s0, s2
24         stur    s0, [x14, #-4]
25         ldr     s0, [x10], #8
26         ldr     s1, [x11], #8
27         fadd    s0, s0, s1
28         ldr     s1, [x12], #8
29         fadd    s0, s0, s1
30         str     s0, [x14], #8
31         b.ne    .LBB0_4
32         cbz     w9, .LBB0_7
33 .LBB0_6:
34         lsl     x8, x8, #2
35         ldr     s0, [x0, x8]
36         ldr     s1, [x1, x8]
```

Loop Unrolling
4 Single-core CPU Optimizations

Initial code without loop unrolling

```c
void basic_loop1(const float *A,
                 const float *B,
                 const float *C,
                 const int n)
{
  #pragma unroll 2
  for (int i = 0; i < n; i++) {
    D[i] = A[i] + B[i] + C[i];
  }
}
```

- Compiler: Clang 10.0.0 – Options:

-O3 -march=armv8-a+nosimd -funroll-loops

Problem: the code becomes much bigger!

```asm
basic_loop1:
        cmp     w4, #1
        b.lt    .LBB0_7
        cmp     w4, #1
        b.ne    .LBB0_3
        mov     x8, xzr
        b       .LBB0_6
.LBB0_3:
        and     w9, w4, #0x1
        mov     x8, xzr
        add     x10, x0, #4
        add     x11, x1, #4
        add     x12, x2, #4
        sub     w13, w4, #1
        add     x14, x3, #4
.LBB0_4:    // =>This Inner Loop Header: Depth=1
        ldur    s0, [x10, #-4]
        ldur    s1, [x11, #-4]
        ldur    s2, [x12, #-4]
        add     x8, x8, #2
        cmp     w13, w8
        fadd    s0, s0, s1
        fadd    s0, s0, s2
        stur    s0, [x14, #-4]
        ldr     s0, [x10], #8
        ldr     s1, [x11], #8
        fadd    s0, s0, s1
        ldr     s1, [x12], #8
        fadd    s0, s0, s1
        str     s0, [x14], #8
        b.ne    .LBB0_4
        cbz     w9, .LBB0_7
.LBB0_6:
        lsl     x8, x8, #2
        ldr     s0, [x0, x8]
        ldr     s1, [x1, x8]
```

Code with loop unrolling

```
1 for (i = 0; i < n; i += 2) {
2   D[i + 0] = A[i + 0] + B[i + 0] * C[i + 0];
3   D[i + 1] = A[i + 1] + B[i + 1] * C[i + 1];
4 }
```

Code with loop unrolling and jam

```
1 for (i = 0; i < n; i += 2) {
2   d0 = A[i + 0] + B[i + 0];
3   d1 = A[i + 1] + B[i + 1];
4   D[i + 0] = d0 * C[i + 0];
5   D[i + 1] = d1 * C[i + 1];
6 }
```

- Breaks data dependencies

- We can start calculating a part of D[i+1] (in the variable d1) while D[i+0], has not yet been completely computed

- This optimization can sometimes be performed by the compiler

- Requires more registers (or memory)

Sliding sum of 3 points :

```
1 void sum(const float *A, float *B, const int n) {
2   for (int i = 1; i < n; i++) {
3     B[i] = A[i - 1] + A[i + 0] + A[i + 1];
4   }
5 }
```

Sliding sum of 3 points with 3rd-order unrolling:

```
1 void sum_u3(const float *A, float *B, const int n) {
2   for (int i = 1; i < n; i += 3) {
3     B[i + 0] = A[i - 1] + A[i + 0] + A[i + 1];
4     B[i + 1] = A[i + 0] + A[i + 1] + A[i + 2];
5     B[i + 2] = A[i + 1] + A[i + 2] + A[i + 3];
6   }
7   // pas d'épilogue pour simplifier
8 }
```

Sliding sum of 3 points with 3rd-order unrolling and variables rotation:

```
1 void sum_u3_rot(const float *A, float *B, const int n) {
2   float a0 = A[0];
3   float a1 = A[1];
4   for (int i = 1; i < n; i += 3) {
5     // only 3 reads into A
6     float a2 = A[i + 1], a3 = A[i + 2], a4 = A[i + 3];
7     B[i + 0] = a0 + a1 + a2;
8     B[i + 1] = a1 + a2 + a3;
9     B[i + 2] = a2 + a3 + a4;
10    // rotation on a0 and a1 variables
11    a0 = a3;
12    a1 = a4;
13  }
14 }
```

```
 1  sum_u3:
 2          cmp     w2, #3
 3          b.lt    .LBB0_3
 4          sub     w10, w2, #1
 5          add     x8, x0, #8
 6          add     x9, x1, #8
 7          sxtw    x10, w10
 8          mov     w11, #1
 9  .LBB0_2: // =>This Inner Loop Header: Depth=1
10          ldp     s0, s1, [x8, #-8]  // <= 2 loads
11          ldr     s2, [x8]           // <= 1 load
12          add     x11, x11, #3
13          cmp     x11, x10
14          fadd    s0, s0, s1
15          fadd    s0, s0, s2
16          stur    s0, [x9, #-4]
17          ldp     s0, s1, [x8, #-4]  // <= 2 loads
18          ldr     s2, [x8, #4]       // <= 1 load
19          fadd    s0, s0, s1
20          fadd    s0, s0, s2
21          str     s0, [x9]
22          ldp     s0, s1, [x8]       // <= 2 loads
23          fadd    s0, s0, s1
24          ldr     s1, [x8, #8]       // <= 1 load
25          add     x8, x8, #12
26          fadd    s0, s0, s1
27          str     s0, [x9, #4]
28          add     x9, x9, #12
29          b.lt    .LBB0_2
30  .LBB0_3:
31          ret
```

```
 1  sum_u3_rot:
 2          cmp     w2, #2
 3          b.lt    .LBB0_3
 4          ldp     s1, s0, [x0]
 5          mov     w8, w2
 6          add     x9, x0, #16
 7          add     x10, x1, #8
 8          mov     w11, #1
 9  .LBB0_2: // =>This Inner Loop Header: Depth=1
10          ldp     s2, s3, [x9, #-8]  // <= 2 loads
11          ldr     s4, [x9], #12      // <= 1 load
12          fadd    s1, s0, s1
13          add     x11, x11, #3
14          fadd    s0, s0, s2
15          fadd    s1, s1, s2
16          fadd    s2, s2, s3
17          stur    s1, [x10, #-4]
18          fadd    s0, s0, s3
19          fadd    s1, s2, s4
20          cmp     x11, x8
21          stp     s0, s1, [x10], #12
22          fmov    s1, s3             // <= register rotation
23          fmov    s0, s4             // <= register rotation
24          b.lo    .LBB0_2
25  .LBB0_3:
26          ret
```

50/68

Variables Rotation & Reduction
4 Single-core CPU Optimizations

```
1 void sum_u3_rot(const float *A, float *B, const int n)
2   float a0 = A[0], a1 = A[1];
3   for (int i = 1; i < n; i += 3) {
4     // only 3 reads into A
5     float a2 = A[i + 1], a3 = A[i + 2], a4 = A[i + 3];
6     B[i + 0] = a0 + a1 + a2; // 2 additions
7     B[i + 1] = a1 + a2 + a3; // additions
8     B[i + 2] = a2 + a3 + a4; // 2 additions
9     // rotation on a0 and a1 variables
10    a0 = a3; a1 = a4;
11  }
12 }
```

THIS CODE IS WRONG!!!!

```
1 void sum_u3_rot_red(const float *A, float *B, const int n)
2 {
3   float a0 = A[0], a1 = A[1], a2 = A[2], a3 = A[3];
4   // sums s0, s1, s2 = reductions
5   float s0 = a0 + a1 + a2;
6   float s1 = a1 + a2 + a3;
7   for (int i = 1; i < n; i += 3) {
8     // only one read into A
9     float a4 = A[i + 4];
10    // compute only 2 additions
11    float s2 = a2 + a3 + a4;
12    B[i + 0] = s0;
13    B[i + 1] = s1;
14    B[i + 2] = s2;
15    // rotation on a0, a1, a2, a3 and a4 variables
16    a0 = a1; a1 = a2; a2 = a3; a3 = a4;
17    // rotation on the s0, s1, s2 variables
18    s0 = s1; s1 = s2;
19  }
20 }
```

- This code compute 6 additions, can we compute less?
  — Yes! Let us use a reduction!

Rot. & Red – Asm
4 Single-core CPU Optimizations

```
1  void sum_u3_rot_red(const float *A, float *B, const int n)
2  {
3    float a0 = A[0], a1 = A[1], a2 = A[2], a3 = A[3];
4    // sums s0, s1, s2 = reductions
5    float s0 = a0 + a1 + a2;
6    float s1 = a1 + a2 + a3;
7    for (int i = 1; i < n; i += 3) {
8      // only one read into A
9      float a4 = A[i + 4];
10     // compute only 2 additions
11     float s2 = a2 + a3 + a4;
12     B[i    ] = s0;
13     B[i + 1] = s1;
14     B[i + 2] = s2;
15     // rotation on a0, a1, a2, a3 and a4 variables
16     a0 = a1; a1 = a2; a2 = a3; a3 = a4;
17     // rotation on the s0, s1, s2 variables
18     s0 = s1; s1 = s2;
19   }
20 }
```

THIS CODE IS WRONG!!!!

```
1  sum_u3_rot_red:
2      cmp     w2, #3
3      b.lt    .LBB0_3
4      ldp     s2, s0, [x0, #4]
5      ldr     s1, [x0, #12]
6      ldr     s3, [x0], #20
7      sub     w9, w2, #1
8      fadd    s4, s2, s0
9      mov     x8, xzr
10     fadd    s2, s3, s2
11     fadd    s3, s4, s1
12     fadd    s2, s2, s0
13     sxtw    x9, w9
14 .LBB0_2: // =>This Inner Loop Header: Depth=1
15     add     x10, x8, #4
16     cmp     x10, x9
17     lsl     x10, x8, #2
18     fadd    s4, s1, s0  // <= 1 addition
19     fmov    s0, s1      // <= register rotation
20     ldr     s1, [x0, x10]
21     add     x10, x1, x10
22     stp     s2, s3, [x10, #4]
23     fmov    s2, s3      // <= register rotation
24     fadd    s3, s4, s1  // <= 1 addition
25     add     x8, x8, #3
26     str     s3, [x10, #12]
27     b.lt    .LBB0_2
28 .LBB0_3:
29     ret
```

Two independent loops:

```
1 for (int i = 0; i < n; i++)
2   D[i] = A[i] + B[i];
3 for (int i = 0; i < n; i++)
4   E[i] = A[i] * C[i];
```

Merging the two loops into one:

```
1 for (int i = 0; i < n; i++) {
2   D[i] = A[i] + B[i];
3   E[i] = A[i] * C[i];
4 }
```

- This improves data reuse
- In the example, the second reading of `A[i]` (line 2) will necessarily be in the caches ➔ **Temporal locality**

- Split a loop in multiple loops

- The reverse operation of loop fusion

- For special reasons (ex.: multi-threading)

- Simplifies or eliminates a dependency by cutting the loop into two parts

- Sometimes pressure on the registers makes it more interesting to have separate loops

- Conditional Branching Instructions (e.g. `if`, `switch`, etc) create bubbles in the processor pipeline

- The pipeline cannot operate at full efficiency

- As far as possible, we should therefore avoid these instructions in the hotspot of the code
  - However, if the branch is mispredicted, we need to wait the pipeline latency ($\approx$ 15 cycles)

Example of bad code on the left and better code on the right:

```
1 for (int i = 0; i < n; i++) {
2   if (i >= 1 && i < n - 1) {
3     switch (i % 4) {
4       case 0: B[i] = A[i] * 0.3333f;
5       case 1: B[i] = A[i] + 1.3333f;
6       case 2: B[i] = A[i] - 0.7555f;
7       case 3: B[i] = A[i] * 1.1111f;
8       default: break;
9     }
10  }
11 }
```

```
1 for (int i = 1; i < n - 1; i += 4) {
2   B[i + 0] = A[i + 0] + 1.3333f;
3   B[i + 1] = A[i + 1] - 0.7555f;
4   B[i + 2] = A[i + 2] * 1.1111f;
5   B[i + 3] = A[i + 3] * 0.3333f;
6 }
```

- `if` (line 2) can be removed by modifying the start and end of the loop

- `switch` (line 3) can be avoided by unrolling the loop at the 4th order

- When code is limited by memory throughput, you have to be very careful about how we access the data

- Memory bandwidth is a major limiting factor in modern architectures
  — There are mechanisms to reduce this problem: use of *pre-fetching* instructions
  — Memory is accessed by line of words (one word = 32-bit)
  — It's interesting to access data that follow one another (**spatial locality**)
  — Reduce the number of RAM accesses and favor cache accesses (**temporal locality**)

```
1 for (int i = 0; i < n; i++) // column
2   for (int j = 0; j < n; j++) // row
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```
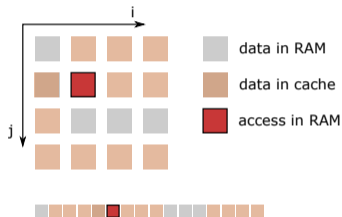

Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int i = 0; i < n; i++) // column
2   for (int j = 0; j < n; j++) // row
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1  for (int i = 0; i < n; i++) // column
2    for (int j = 0; j < n; j++) // row
3      C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int i = 0; i < n; i++) // column
2   for (int j = 0; j < n; j++) // row
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int i = 0; i < n; i++) // column
2   for (int j = 0; j < n; j++) // row
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int i = 0; i < n; i++) // column
2   for (int j = 0; j < n; j++) // row
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int i = 0; i < n; i++) // column
2   for (int j = 0; j < n; j++) // row
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

- In this implementation, data accesses are not contiguous in memory
- There is a stride of 4 words between each access (not good for spatial locality)

```
1 for (int j = 0; j < n; j++) // row
2   for (int i = 0; i < n; i++) // column
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```
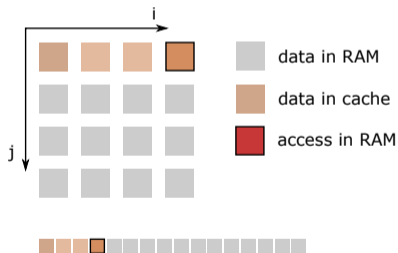

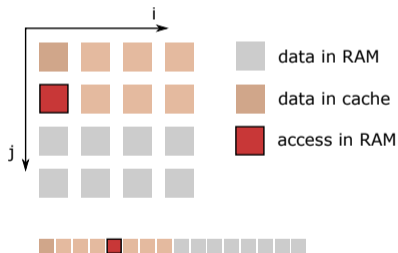Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int j = 0; j < n; j++) // row
2   for (int i = 0; i < n; i++) // column
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int j = 0; j < n; j++) // row
2   for (int i = 0; i < n; i++) // column
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```
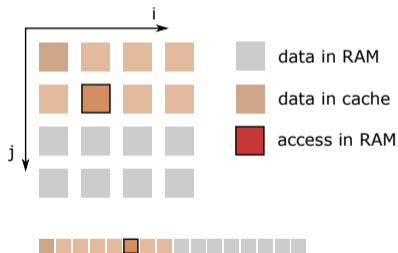

Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1  for (int j = 0; j < n; j++) // row
2    for (int i = 0; i < n; i++) // column
3      C[j * n + i] = A[j * n + i] + B[j * n + i];
```
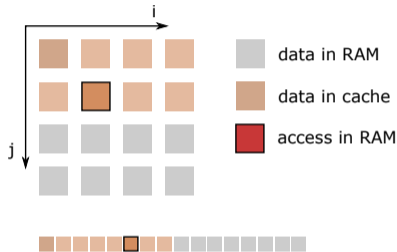

Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int j = 0; j < n; j++) // row
2   for (int i = 0; i < n; i++) // column
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int j = 0; j < n; j++) // row
2   for (int i = 0; i < n; i++) // column
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

```
1 for (int j = 0; j < n; j++) // row
2   for (int i = 0; i < n; i++) // column
3     C[j * n + i] = A[j * n + i] + B[j * n + i];
```


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)

Logical and hardware view of memory accesses

- In this implementation, accesses are contiguous in memory
  — Cache lines are fully used
  — Memory throughput is maximized

- i-loop and j-loop have simply been switched

Cache Blocking
4 Single-core CPU Optimizations

- In many cases, data can be reused (spatial locality)

- Let us take the example of a *stencil* code operating on a 2D grid

```
1 for (int j = 1; j < rows - 1; j++) // row
2   for (int i = 1; i < cols - 1; i++) // column
3     B[j * cols + i] = A[(j  ) * cols + (i-1)] + A[(j  ) * cols + (i+1)] + // left, right
4                       A[(j  ) * cols + (i  )] +                           // center
5                       A[(j-1) * cols + (i  )] + A[(j+1) * cols + (i  )]; // top, bottom
```
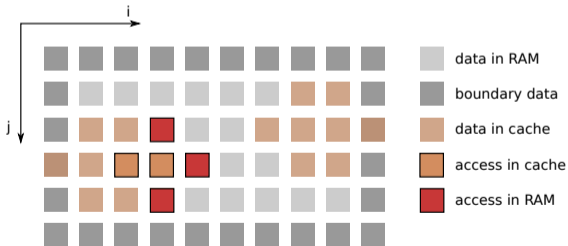
```
1 for (int j = 1; j < rows - 1; j++) // row
2   for (int i = 1; i < cols - 1; i++) // column
3     B[j * cols + i] = A[(j  ) * cols + (i-1)] + A[(j  ) * cols + (i+1)] + // left, right
4                       A[(j  ) * cols + (i  )] +                          // center
5                       A[(j-1) * cols + (i  )] + A[(j+1) * cols + (i  )]; // top, bottom
```
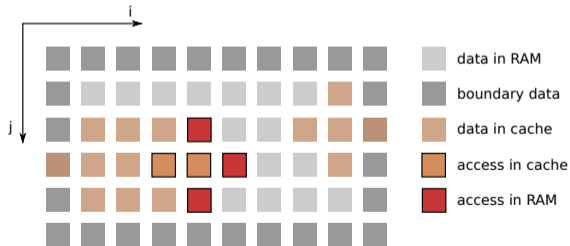

Logical view of the 2D grid

WITHOUT Cache Blocking – Part 1
4 Single-core CPU Optimizations

```
1 for (int j = 1; j < rows - 1; j++) // row
2   for (int i = 1; i < cols - 1; i++) // column
3     B[j * cols + i] = A[(j  ) * cols + (i-1)] + A[(j  ) * cols + (i+1)] + // left, right
4                       A[(j  ) * cols + (i  )] +                           // center
5                       A[(j-1) * cols + (i  )] + A[(j+1) * cols + (i  )];  // top, bottom
```
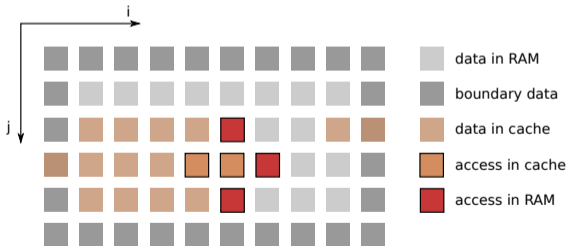

Logical view of the 2D grid

WITHOUT Cache Blocking – Part 1

4 Single-core CPU Optimizations

```
1 for (int j = 1; j < rows - 1; j++) // row
2   for (int i = 1; i < cols - 1; i++) // column
3     B[j * cols + i] = A[(j  ) * cols + (i-1)] + A[(j  ) * cols + (i+1)] + // left, right
4                       A[(j  ) * cols + (i  )] +                          // center
5                       A[(j-1) * cols + (i  )] + A[(j+1) * cols + (i  )]; // top, bottom
```
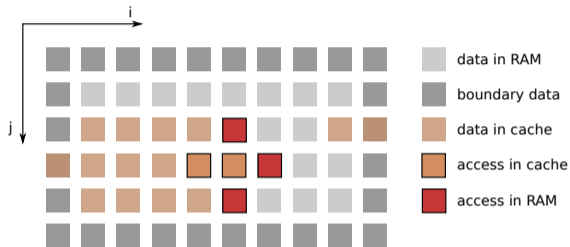

Logical view of the 2D grid

```
1 for (int j = 1; j < rows - 1; j++) // row
2   for (int i = 1; i < cols - 1; i++) // column
3     B[j * cols + i] = A[(j  ) * cols + (i-1)] + A[(j  ) * cols + (i+1)] + // left, right
4                       A[(j  ) * cols + (i  )] +                          // center
5                       A[(j-1) * cols + (i  )] + A[(j+1) * cols + (i  )]; // top, bottom
```


Logical view of the 2D grid

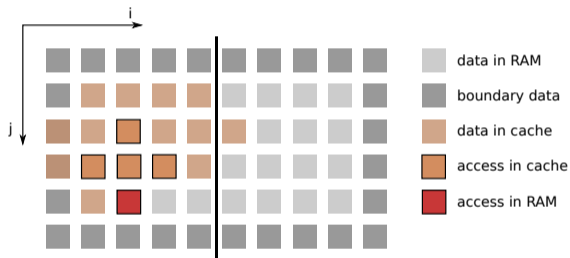# WITHOUT Cache Blocking – Part 2
4 Single-core CPU Optimizations

Logical 2D grid memory view

- For each increment of i there are 3 new RAM accesses and 2 cache accesses (we neglect cache lines)
- Can we reduce the number of RAM accesses?
  — Yes, using a so-called *cache blocking* technique (also called *tiling* in the literature)
  — The idea is to modify the data path to maximize reuses

Logical 2D grid memory view

Logical 2D grid memory view

Logical 2D grid memory view

Logical 2D grid memory view

Logical 2D grid memory view

	data in RAM
	boundary data
	data in cache
	access in cache
	access in RAM

Logical 2D grid memory view

- Cache blocking reduces the number of RAM accesses
  — On average, only one RAM access per point remains!
  — We have divided the grid into several blocks (here 2), vertical blocks

Cache Blocking – Blocks Size

- The block size depends on the problem and on the CPU architecture
- For the previous stencil code, the size of a block can be defined as follows:

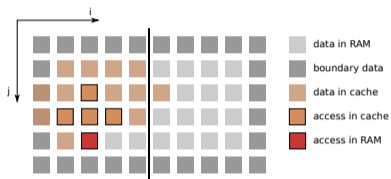$$blockSize = \frac{sizeOfCache}{2 \times 3 \times sizeOfData},$$

with $sizeOfCache$ the size of the L3 cache in bytes and $sizeOfData$ the size of the data (single precision = 4 bytes, double precision = 8 bytes).

- We divide by 2 because caches generally work better when half is used (grandma's recipe...)
- We divide by 3 because we need to keep 3 lines cached in our stencil
- Note that if $blockSize \geq cols$ then the cache blocking technique is useless

Cache Blocking – Implementation
4 Single-core CPU Optimizations

data in RAM
boundary data
data in cache
access in cache
access in RAM

```c
1 #define SIZE_OF_CACHE_L3 96 // we suppose a L3 cache of 96 bytes for the example
2 int blockSize = SIZE_OF_CACHE_L3 / (2 * 3 * sizeof(float)); // (96 / 24) = 4
3
4 for (int iOff = 1; iOff < cols - 1; iOff += blockSize) { // loop over vertical blocks
5   blockSize = min(cols - 1 - iOff, blockSize); // reduce the block size if needed
6   for (int j = 1; j < rows - 1; j++) // row
7     for (int i = iOff; i < iOff + blockSize; i++) // column
8       B[j * cols + i] = A[(j   ) * cols + (i - 1)] + A[(j   ) * cols + (i + 1)] +
9                         A[(j   ) * cols + (i   )] +
10                        A[(j - 1) * cols + (i   )] + A[(j + 1) * cols + (i   )];
11 }
```

Q&A

Thank you for listening!
Do you have any questions?