

# Introduction to High Performance Computing

## Focus on Single-core CPU

Polytech Sorbonne – EI5-SE – Calcul haute performance (EPU-F9-IHP)

Adrien CASSAGNE

September 29, 2025



# Table of Contents

1 Introduction

- ▶ Introduction
- ▶ High Performance Computing
- ▶ Programmable Architectures
- ▶ Single-core CPU Architecture
- ▶ Single-core CPU Optimizations



# Lecturer

## 1 Introduction

Adrien CASSAGNE

- Associate Professor in Computer Science at Sorbonne University
- Research: Laboratoire d'Informatique de Sorbonne Université (LIP6)
  - Hardware and software for embedded system team (ALSoC)
  - Computer Vision on embedded heterogeneous SoCs
  - Software-Defined Radio on multi-core & SIMD CPUs
  - Computational Fluid Dynamics on Supercomputers (CPUs & discrete GPUs)
- E-mail : `adrien.cassagne [at] sorbonne-universite [dot] fr`



# Contents of the Class

## 1 Introduction

- $7 \times 2$  hours of classroom lectures
  - Session 1: Introduction to High Performance Computing and Dalek
  - Session 2: Optimizations for Single-core CPUs
  - Session 3: Single Instruction Multiple Data (SIMD) Model on CPUs
  - Session 4: Multi-threaded Programming on CPUs (part 1)
  - Session 5: Multi-threaded Programming on CPUs (part 2)
  - Session 6: Multi-node Programming on CPUs
  - Session 7: GPU Programming Model
- $7 \times 2$  hours of hands-on sessions
  - Based on the EasyPAP framework
- $5 \times 4$  hours for the projects
  - **Projects defense: TBD**



# Aims of this Course

## 1 Introduction

In General:

- Understanding better today's architectures
- Learn about the different parallelism models available
- Introduction to code optimization
- Introduction to parallel programming

Today:

- What is high-performance computing?
- Overview of programmable architectures



# Table of Contents

2 High Performance Computing

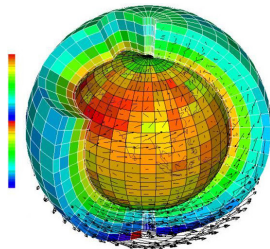
- ▶ Introduction
- ▶ High Performance Computing
- ▶ Programmable Architectures
- ▶ Single-core CPU Architecture
- ▶ Single-core CPU Optimizations



# High Performance Computing, québécois?

## 2 High Performance Computing

- Association of a large number of processors (thousands or even millions) in the broadest sense (CPU, GPU, FPGA, etc.)
- Building massively parallel architectures
- Solving computationally-intensive problems
  - Modelling climate change
  - Optimizing energy production
  - Subsoil analysis for oil exploration
  - Code cracking in cryptography
  - Financial analysis
  - ...
- Major challenges for companies and governments alike





# Supercomputers

## 2 High Performance Computing

- Physical/hardware support for high performance computing
- 1 petaflop/s =  $10^{15}$  floating-point operations per second



Adastra Supercomputers @ CINES (Montpellier), 61 PFlops/s, top 20.





# Top 500: The most Powerful Supercomputers

2 High Performance Computing

Top 500 in June 2022: <https://www.top500.org/lists/top500/2022/06/>.

Rank	System Name	Country	# Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier	USA	8,730,112	1,102.00	1,685.65	21,100
2	Fugaku	Japan	7,630,848	442.01	537.21	29,899
3	LUMI	Finland	1,110,144	151.90	214.35	2,942
4	Summit	USA	2,414,592	148.60	200.79	10,096
5	Sierra	USA	1,572,480	94.64	125.71	7,438
6	Sunway TaihuLight	China	10,649,600	93.01	125.44	15,371
7	Perlmutter	USA	761,856	70.87	93.75	2,589
8	Selene	USA	555,520	63.46	79.22	2,646
9	Tianhe-2A	China	4,981,760	61.44	100.68	18,482
10	Adastra	France	319,072	46.10	61.61	921



# Top 500: The most Powerful Supercomputers

2 High Performance Computing

Top 500 in June 2024: <https://www.top500.org/lists/top500/2024/06/>.

Rank	System Name	Country	# Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier	USA	8,699,904	1,206.00	1,714.81	22,786
2	Aurora	USA	9,264,128	1,012.00	1,980.01	38,698
3	Microsoft Eagle	USA	2,073,600	561.20	846.84	–
4	Fugaku	Japan	7,630,848	442.01	537.21	29,899
5	LUMI	Finland	2,752,704	379.70	531.51	7,107
6	Alps	Switzerland	1,305,600	270.00	353.75	5,194
7	Leonardo	Italy	1,824,768	241.20	306.31	7,494
8	MareNostrum 5	Spain	663,040	175.30	249.44	4,159
9	Summit	USA	2,414,592	148.60	200.79	10,096
10	NVIDIA Eos DGX	USA	485,888	121.40	188.65	–



# Top 500: The most Powerful Supercomputers

2 High Performance Computing

Top 500 in June 2025: <https://top500.org/lists/top500/2025/06/>.

Rank	System Name	Country	# Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	El Capitan	USA	11,039,616	1,742.00	2,746.38	29,581
2	Frontier	USA	9,066,176	1,353.00	2,055.72	24,607
3	Aurora	USA	9,264,128	1,012.00	1,980.01	38,698
4	JUPITER Booster	Germany	4,801,344	793.40	930.00	13,088
5	Microsoft Eagle	USA	2,073,600	561.20	846.84	–
6	HPC6	Italy	3,143,520	477.90	606.97	8,461
7	Fugaku	Japan	7,630,848	442.01	537.21	29,899
8	Alps	Switzerland	2,121,600	434.90	574.84	7,124
9	LUMI	Finland	2,752,704	379.70	531.51	7,107
10	Leonardo	Italy	1,824,768	241.20	306.31	7,494



# Nodes Architecture in 2025

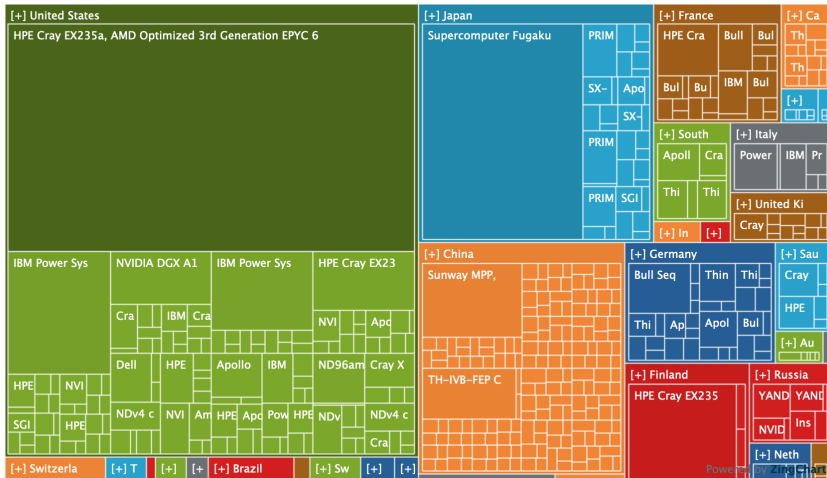
## 2 High Performance Computing

- Top 1: AMD Instinct APU (CPU+GPU)
  - Top 2, 6 and 9: AMD EPYC CPU + AMD Instinct GPU
  - Top 3: Intel Xeon CPU + Intel Data Center GPU
  - Top 4 and 8: Nvidia GH200 Superchip (CPU+GPU)
  - Top 5 and 10: Intel Xeon CPU + Nvidia H100/A100 GPU
  - Top 7: Fujitsu A64FX CPU (ARM + SVE)
- 
- Mostly x86\_64 CPUs with AVX-512, some ARM CPUs with SVE
  - Mostly AMD and Nvidia GPUs, few Intel GPUs



# Top 500 (June 2022): Tree Map

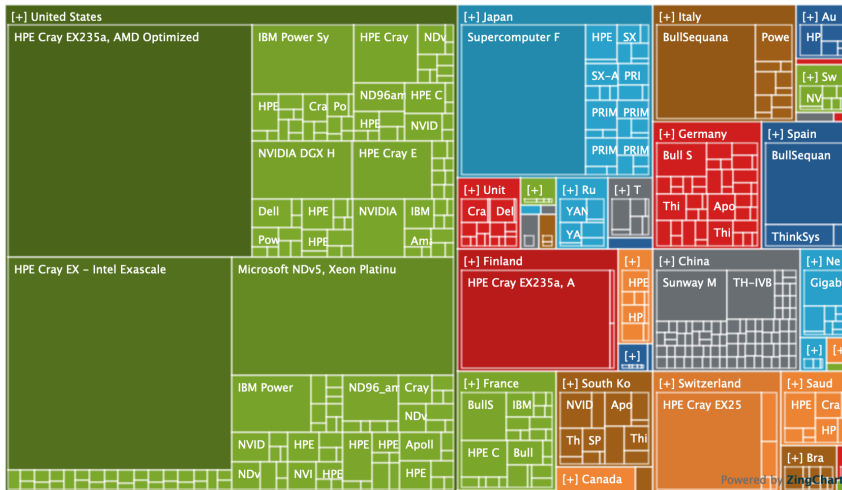
2 High Performance Computing





# Top 500 (June 2024): Tree Map

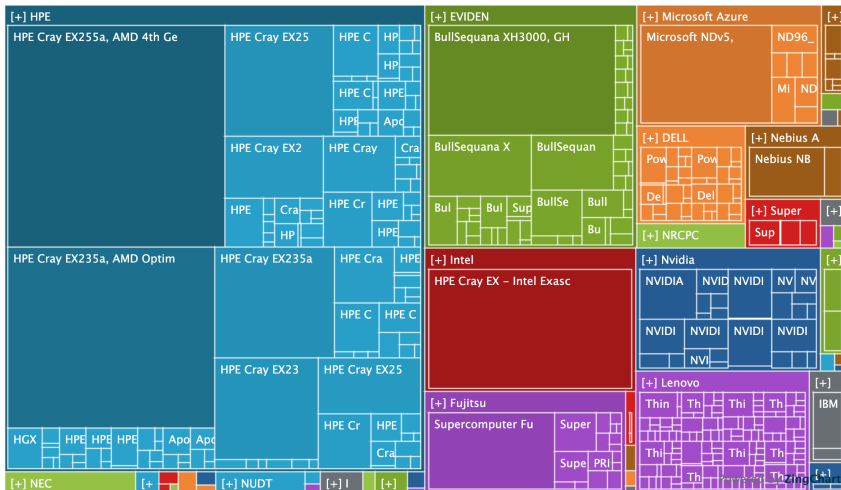
2 High Performance Computing





# Top 500 (June 2025): Tree Map

2 High Performance Computing





## And what?

### 2 High Performance Computing

- No access to a machine of this type unfortunately
- An assembly of a large number ( $> 1000$ ) of computers
  - Around 100 cores per CPU or more
  - Fiber-optic network (InfiniBand)
  - Cloud-like storage space and even faster
- Approximations for this course
  - **If you can program a small cluster “correctly”, then you’re not far from being able to program a supercomputer.**
  - To be put into perspective, since we won’t be seeing (or only a few) problems related to data transfers between different nodes (node = computer).
- Also valid for
  - Programming embedded and real-time systems
  - Compute-intensive software (video games, image processing, etc.)





# Table of Contents

3 Programmable Architectures

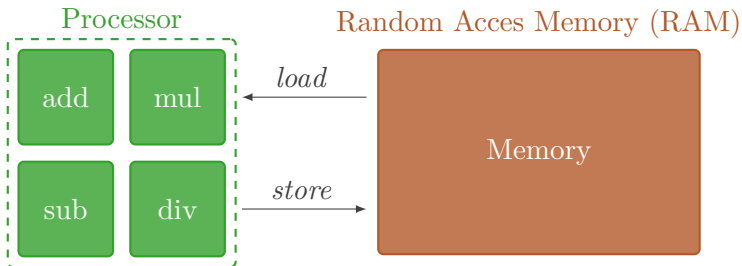
- ▶ Introduction
- ▶ High Performance Computing
- ▶ Programmable Architectures
- ▶ Single-core CPU Architecture
- ▶ Single-core CPU Optimizations



# Simplified CPU Architecture – Part 1

3 Programmable Architectures

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

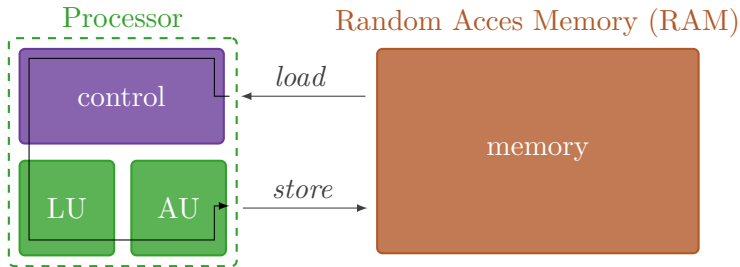




## Simplified CPU Architecture – Part 2

3 Programmable Architectures

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





# CPU Arch: Cycle, Freq. and Throughput

## 3 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)
- Modern CPU/RAM frequency: between 1 GHz and 4 GHz ( $1 \text{ GHz} = 10^9 \text{ Hz}$ )
- Memory throughput (RAM)  $\approx 50 \text{ GB/s}$  (DDR5)
- CPU capable of consuming/producing data at  $\approx 2 \text{ TB/s}$ 
  - AMD Ryzen 7 7840U: CPU inside the ASUS ROG Ally
  - **The CPU is much faster** ( $2000/50 = \times 40$ ) !
  - How can we solve this problem?



# CPU Architecture: Memory Hierarchy – Part 1

3 Programmable Architectures

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

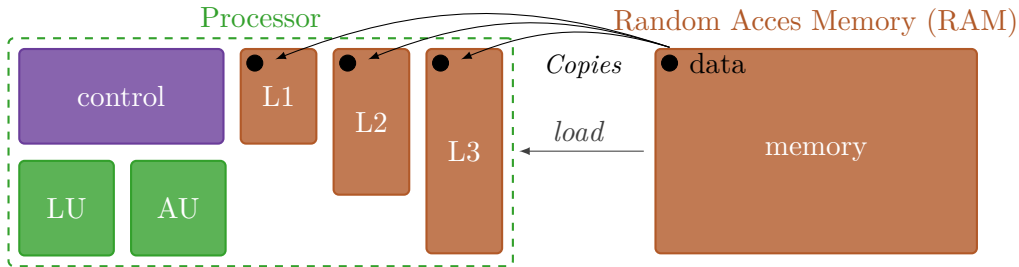
- Faster memory between CPU and RAM = **Cache memory**
  - Faster memory is more expensive and takes up physical space
  - This memory (= cache) is much smaller than RAM
  - 3 levels of cache (in the processor):
    - L1, the fastest and smallest (32 KB), access time  $\approx 2$  cycles
    - L2, slower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



# CPU Architecture: Memory Hierarchy – Part 2

3 Programmable Architectures

Scenario: **first** *load* of a data from memory



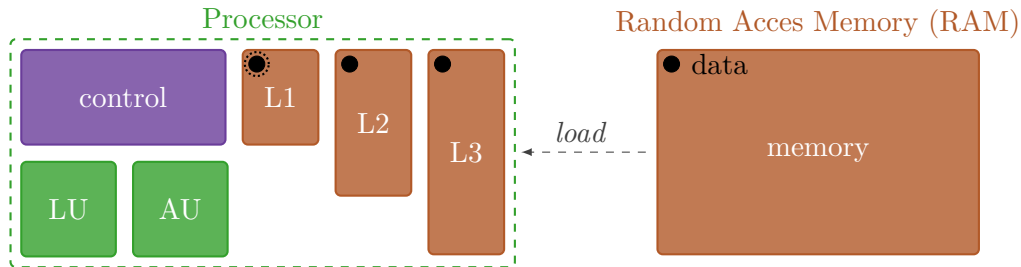
Loading data from RAM:  $\approx 100$  cycles



# CPU Architecture: Memory Hierarchy – Part 3

3 Programmable Architectures

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



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

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



# CPU Architecture: Memory Hierarchy – Part 4

3 Programmable Architectures

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)
- Some architectures implement **non-temporal stores** (or streaming stores)
  - Data is not copied in the caches
  - Avoid cache latency and pollution
  - Example: `VMOVNTDQ` instruction in x86





# CPU Architecture: Spatial Locality

3 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!**



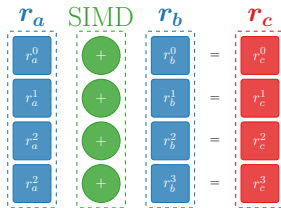
# CPU Architecture: SIMD Instructions

## 3 Programmable Architectures

- Scalar instruction: produces data during 1 cycle



- SIMD instruction: produces  $n$  data during 1 cycle



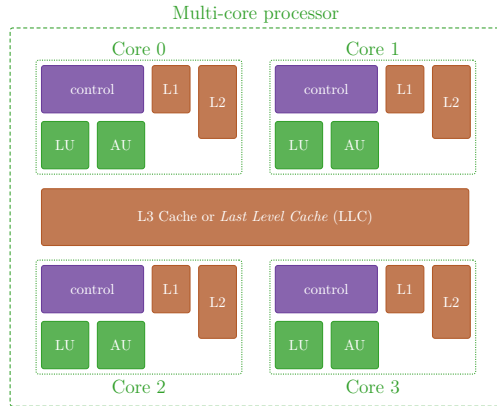
- SIMD instructions operate on so-called “vector registers”.



# CPU Architecture: Multi-core

3 Programmable Architectures

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

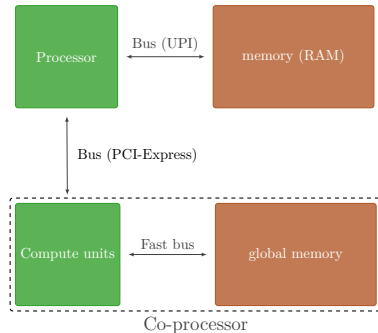




# Co-processors

## 3 Programmable Architectures

- 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

## 3 Programmable Architectures

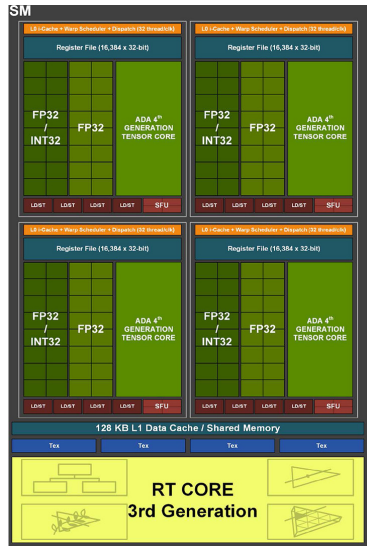
- Originally designed for image processing
- Massively parallel architecture
- Fewer control units than CPUs, but more compute units
- Global memory faster than CPU RAM ( $\approx 500$  GB/s)
- Performance/power consumption ratio generally better than CPUs
- Often suitable for scientific computing
-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



# GPU Architecture: Nvidia Ada Lovelace (1)

## 3 Programmable Architectures

- A *Streaming Multiprocessor* (SM)  $\approx$  a CPU core
- 128 floating-point operations in one cycle 32-bit  
(= 128 CUDA cores)
- Specialized units
  - AI (deep learning): Tensor core
  - Ray tracing: RT core





# GPU Architecture: Nvidia Ada Lovelace (2)

## 3 Programmable Architectures

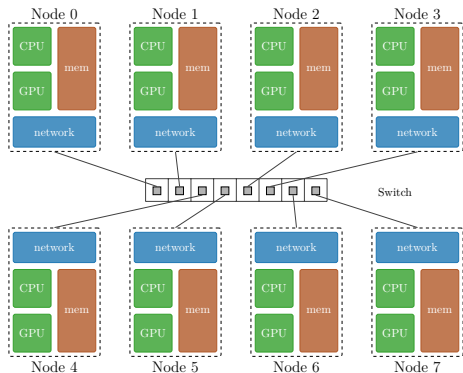


Nvidia GA102 (GeForce RTX 4090), 132 SM and “just” 16 384 CUDA cores.



# Supercomputer Architecture

## 3 Programmable Architectures



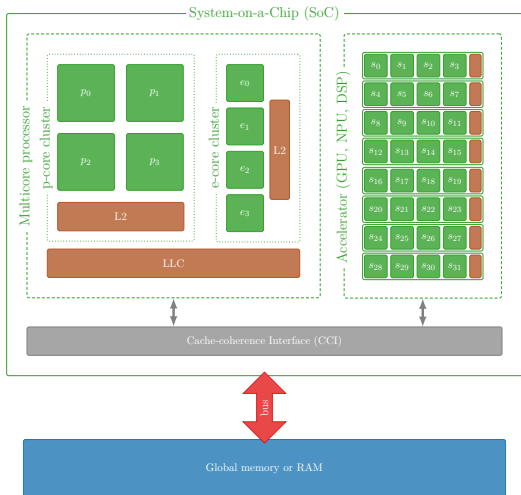
- A very simplistic supercomputer
- Interconnecting computers with a star network (switch)
- Theoretical maximum performance is 8 times that of a node





# Multicore & Heterogeneous SoCs

## 3 Programmable Architectures



- Complex systems
  - Powerful and power efficient CPU cores
  - Specialized process units
  - Various sizes and types of on-chip memory
- Great opportunities
  - Unified global memory (shared pages)
  - Powerful, specialized and low power systems
- Everywhere
  - Nvidia Jetson SBC, Apple M Series
  - AMD APU (PS4-5, Xbox, ...)
  - Qualcomm Snapdragon (smartphones)
  - Intel with iGPU & NPU (Core Ultra)



# Table of Contents

## 4 Single-core CPU Architecture

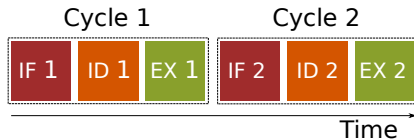
- ▶ Introduction
- ▶ High Performance Computing
- ▶ Programmable Architectures
- ▶ Single-core CPU Architecture
- ▶ Single-core CPU Optimizations



# Pipeless Processor

4 Single-core CPU Architecture

- 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:



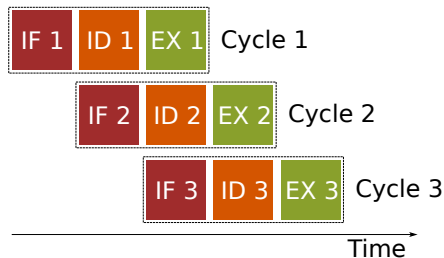
- Cycle 1 and cycle 2 take 6 units of time to complete



# Pipelined Processor – Part 1

## 4 Single-core CPU Architecture

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



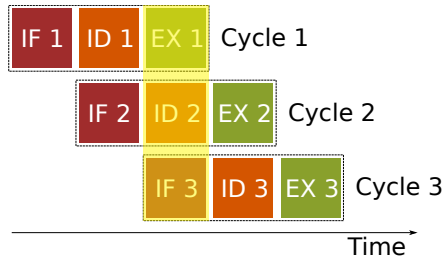
- Cycle 1, cycle 2 and cycle 3 take 5 units of time to complete
- 3-stage pipeline (IF, ID and EX can run in parallel)



## Pipelined Processor – Part 2

### 4 Single-core CPU Architecture

- 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 can deliver 3 times more instructions per cycle
  - Improve the throughput of the CPU
- **Yellow**: the time point when the pipeline is optimal





# Pipelined Processor – Part 3

4 Single-core CPU Architecture

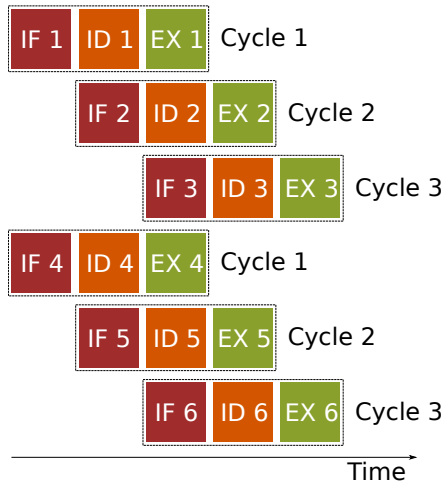
- Today's processors have pipelines of 10 to 20 stages, but the principle remains the same
- There is a mechanism so that the preceding instruction can directly pass its result to the following instruction (*register latch + forwarding*)
- Pipeline seems effective, but what happens if you have conditions (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...



# Super-scalar Processor

## 4 Single-core CPU Architecture

- 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





# Out-of-Order (OoO) Execution

## 4 Single-core CPU Architecture

- Most processors are now capable of executing instructions out of order (different order than the one given by the assembly 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
  - Processor are able to reorder the instructions depending on a fixed **window**
- Here's a code example to illustrate:

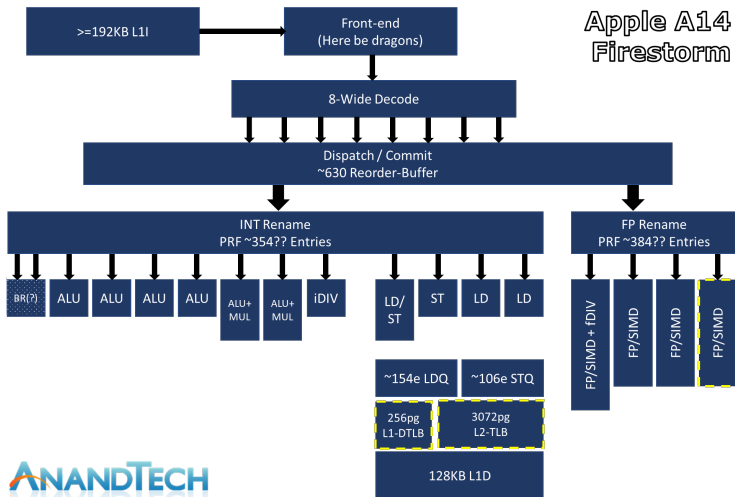
```
1 int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h = 8;  
2 c = a + b; // 1st instruction to be executed, it depends on nothing  
3 e = c * d; // 3rd instruction to be executed, it depends on 'c'  
4 h = f * g; // 2nd instruction to be executed, it depends on nothing
```





# Apple Silicon M1 Micro-architecture (2020)

## 4 Single-core CPU Architecture





# Intel Alder Lake Micro-architecture (2021)

## 4 Single-core CPU Architecture





# Intel Alder Lake Micro-architecture (2021)

## 4 Single-core CPU Architecture

### Front-End

Fetch instructions and decodes them into  $\mu$ ops

#### Large Code

- 128→256 4K iTLB, 16→32 2M/4M iTLB
- Enhanced code prefetch
- 5K→12K branch targets

#### Smarter

Improved branch prediction accuracy  
Smarter code prefetch mechanism

#### Wider

- 16B→32B length decode
- 4→6 decoders
- 6→8  $\mu$ op/cyc from  $\mu$ op\$

#### $\mu$ op\$

- 2.25K→4K  $\mu$ ops:  
increased hit-rate
- increased Frontend BW

#### $\mu$ op Queue

- 70 → 72 entries per thread
- 70 → 144 single thread





## Intel Alder Lake Micro-architecture (2021)

## 4 Single-core CPU Architecture

## Out of Order Engine

Track  $\mu$ op dependencies and dispatch ready  $\mu$ ops to execution units

## Wider

5 → 6 wide allocation  
10 → 12 execution ports

## Deeper

## 512-entry Reorder-Buffer and larger Scheduler sizes

## Smarter

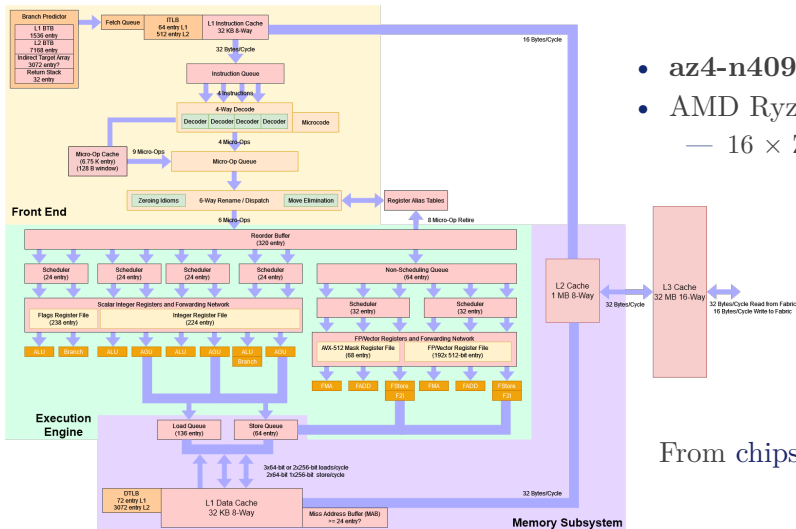
More instructions "executed" at  
rename / allocation stage





# AMD Zen 4 Micro-architecture (2022)

## 4 Single-core CPU Architecture



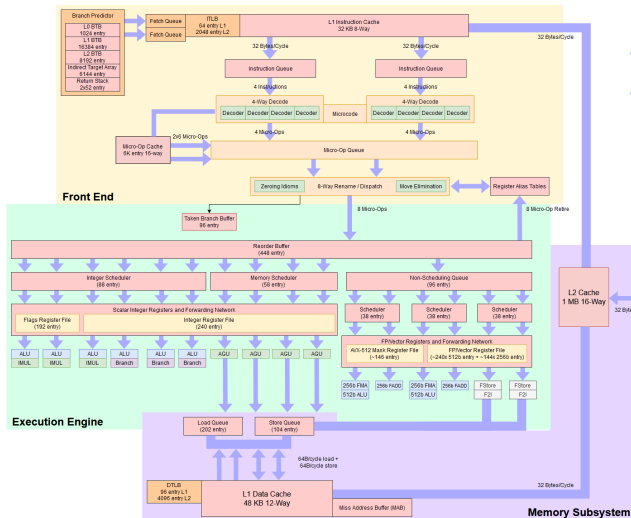
- az4-n4090 and az4-a7900 partitions
- AMD Ryzen 9 7945HX
  - 16 × Zen 4 cores (2-way SMT)

From [chipsandcheese.com](https://chipsandcheese.com)



## 4 Single-core CPU Architecture

## AMD Zen 5



- **az5-a890m** partition
- AMD Ryzen AI 9 HX 370
  - $4 \times$  Zen 5 cores (2-way SMT)
  - $8 \times$  Zen 5c cores (2-way SMT)

From [chipsandcheese.com](http://chipsandcheese.com)

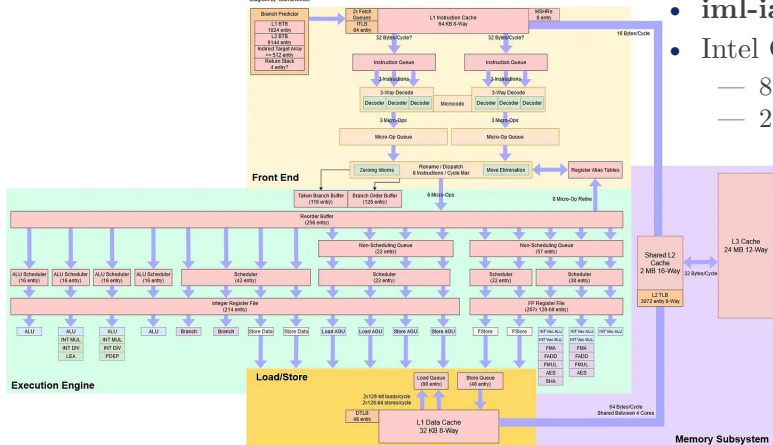


# Intel Crestmont Micro-architecture (2023)

## 4 Single-core CPU Architecture

### Crestmont

Diagrams By: ChrisHouder



- iml-ia770 partition
- Intel Core Ultra 9 185H
  - 8 × Crestmont e-cores (1-way SMT)
  - 2 × Crestmont LPe-c. (1-way SMT)

From [chipsandcheese.com](https://chipsandcheese.com)

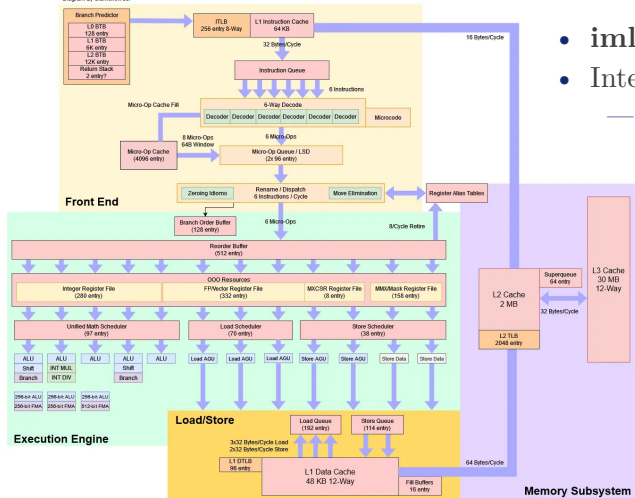


# Intel Redwood Cove Micro-architecture (2023)

## 4 Single-core CPU Architecture

### Redwood Cove

Diagram By Clamchowder



- iml-ia770 partition
- Intel Core Ultra 9 185H
  - 6 × Redwood Cove p-cores (2-way SMT)

From [chipsandcheese.com](https://chipsandcheese.com)





# Table of Contents

5 Single-core CPU Optimizations

- ▶ Introduction
- ▶ High Performance Computing
- ▶ Programmable Architectures
- ▶ Single-core CPU Architecture
- ▶ Single-core CPU Optimizations



# Working with the Compiler

## 5 Single-core CPU Optimizations

- 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

## 5 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>



# Compilateur : quelques options spécifiques

## 5 Single-core CPU Optimizations

- `-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

## 5 Single-core CPU Optimizations

- 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 11 cycles, throughput of 3 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 6 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



# Division Example on *Intel Skylake* Architecture

## 5 Single-core CPU Optimizations

```
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; // 11 cycles for the first time, and 3 cycles after  
4 }
```

- div1 – Number of theoretical cycles:  $11 + (n - 1) \times 3$

```
1 int div2(const float *A, float *B, const int n) {  
2     float inv3 = 1.f / 3.f; // 11 cycles  
3     for (int i = 0; i < n; i++)  
4         B[i] = A[i] * inv3; // 4 cycles for the first time, and 0.5 cycle after  
5 }
```

- div2 – Number of theoretical cycles:  $11 + 4 + (n - 1) \times 0.5$



# Division Example – ARMv8-A Assembly Code

5 Single-core CPU Optimizations

- 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, #3.000000000
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

## 5 Single-core CPU Optimizations

- 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





# Function Calls

## 5 Single-core CPU Optimizations

- 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); // here we call this function very often  
8 }
```



# Inlining

## 5 Single-core CPU Optimizations

- 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

## 5 Single-core CPU Optimizations

- Loop unrolling 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



# Loop Unrolling – Example

5 Single-core CPU Optimizations

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 2<sup>nd</sup>-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 need 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

## 5 Single-core CPU Optimizations

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     #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

```
1 basic_loop1:  
2     cmp     w4, #1  
3     b.lt    .LBBO_7  
4     cmp     w4, #1  
5     b.ne    .LBBO_3  
6     mov     x8, xzr  
7     b       .LBBO_6  
8 .LBBO_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 .LBBO_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    .LBBO_4  
32    cbz     w9, .LBBO_7  
33 .LBBO_6:  
34    lsl     x8, x8, #2  
35    ldr     s0, [x0, x8]  
36    ldr     s1, [x1, x8]
```



# Unroll & Jam

## 5 Single-core CPU Optimizations

### 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 (**good for ILP**)
- 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)





# Variables Rotation

## 5 Single-core CPU Optimizations

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 3<sup>rd</sup>-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     // no epilogue to simplify the code
8 }
```

Sliding sum of 3 points with 3<sup>rd</sup>-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

```



# Loop Fusion

5 Single-core CPU Optimizations

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**



# Loop Splitting

5 Single-core CPU Optimizations

- 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

## 5 Single-core CPU Optimizations

- 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 need to avoid branching in the hotspot of the code**
  - If the branch is mispredicted, we need to wait the pipeline latency ( $\approx 15$  cycles)



# Conditional Branching Instructions – Example

## 5 Single-core CPU Optimizations

Example of a “bad code” on the left and a “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; break;  
5       case 1: B[i] = A[i] + 1.3333f; break;  
6       case 2: B[i] = A[i] - 0.7555f; break;  
7       case 3: B[i] = A[i] * 1.1111f; break;  
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 }
```

- L2: if can be removed by modifying the start and end of the loop
- L3: switch can be avoided by unrolling the loop at the 4<sup>th</sup> order



# Memory Accesses

## 5 Single-core CPU Optimizations

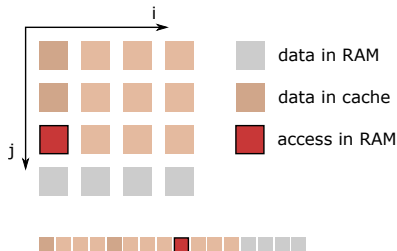
- When code is limited by the memory throughput, we have to be very careful about how we access the data
- **Memory bandwidth is the main limiting factor in modern architectures**
  - There are mechanisms to reduce this problem: use of *pre-fetching* instructions
  - Memory is accessed by line of words (a 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**)



# Example of Memory Accesses – Part 1

5 Single-core CPU Optimizations

```
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];
```



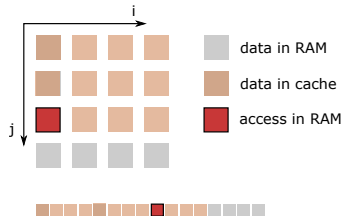
Logical and hardware view of memory accesses (12-word cache & 4-word cache line)





## Example of Memory Accesses – Part 2

5 Single-core CPU Optimizations



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

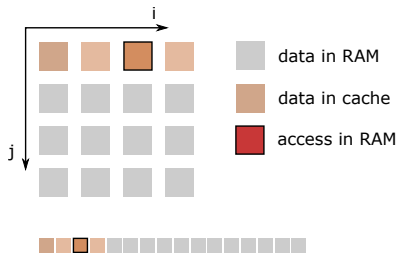
- In this implementation, data accesses are not contiguous in memory
- There is a stride of 4 words between each access (**not good for spatial locality**)



## Example of Memory Accesses – Part 3

5 Single-core CPU Optimizations

```
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];
```

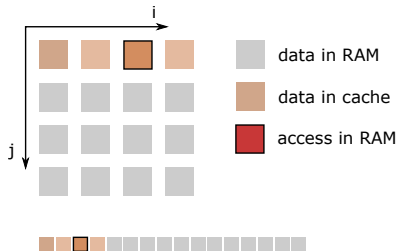


Logical and hardware view of memory accesses (12-word cache & 4-word cache line)



## Example of Memory Accesses – Part 4

5 Single-core CPU Optimizations



Logical and hardware view of memory accesses

- In this implementation, accesses are contiguous in memory
  - Cache lines are fully used (**good for spatial locality**)
  - Memory throughput is maximized
- i-loop and j-loop have simply been switched



# Cache Blocking

## 5 Single-core CPU Optimizations

- In many cases, data can be reused
- Let us take the example of a *stencil* code operating on a 2D grid

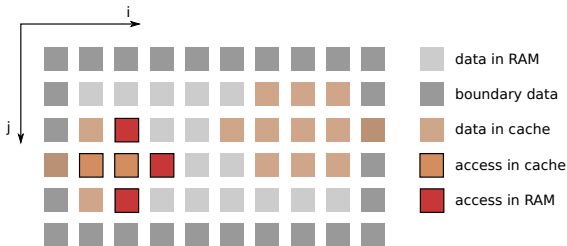
```
1 for (int j = 1; j < rows - 1; j++) // row
2   for (int i = 1; i < cols - 1; i++) // column
3     B[j * cols + i] = A[(j ) * cols + (i-1)] + A[(j ) * cols + (i+1)] + // left, right
4                       A[(j ) * cols + (i )] + // center
5                       A[(j-1) * cols + (i )] + A[(j+1)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



# WITHOUT Cache Blocking – Part 1

## 5 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
```

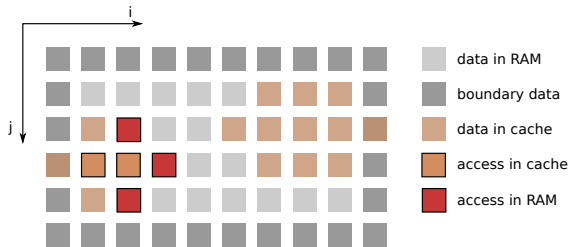


Logical view of the 2D grid



## WITHOUT Cache Blocking – Part 2

### 5 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



- 73/75



## Cache Blocking – Blocks Size

### 5 Single-core CPU Optimizations

- 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).

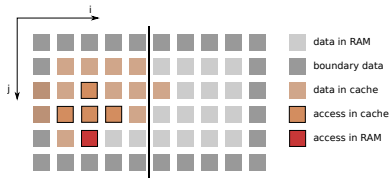
- We divide by 2 because caches generally work better when half is used (grandma's recipe...)
- We divide by 3 because we need to keep 3 lines cached in our stencil
- Note that if  $blockSize \geq cols$  then the cache blocking technique is useless





# Cache Blocking – Implementation

## 5 Single-core CPU Optimizations



```
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?*