The INTERNATIONAL CONFERENCE for HIGH PERFORMANCE COMPUTING NOV 15–20 CHICAGO, IL NETWORKING, STORAGE, & ANALYSIS

SPONSORED BY IEEE COMPUTER SOCIETY TCHPC ACM Association for Computing Machinery sighpc

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

NHR FAU Erlangen National High Performance Computing Center



SC26 Chicago, IL | hpc unites.

Core-Level Performance Engineering

Jan Laukemann[1], Georg Hager[1], Filippo Spiga[2]

[1]Erlangen National High Performance Computing Center (NHR@FAU)

[2]NVIDIA

https://go-nhr.de/CLPE

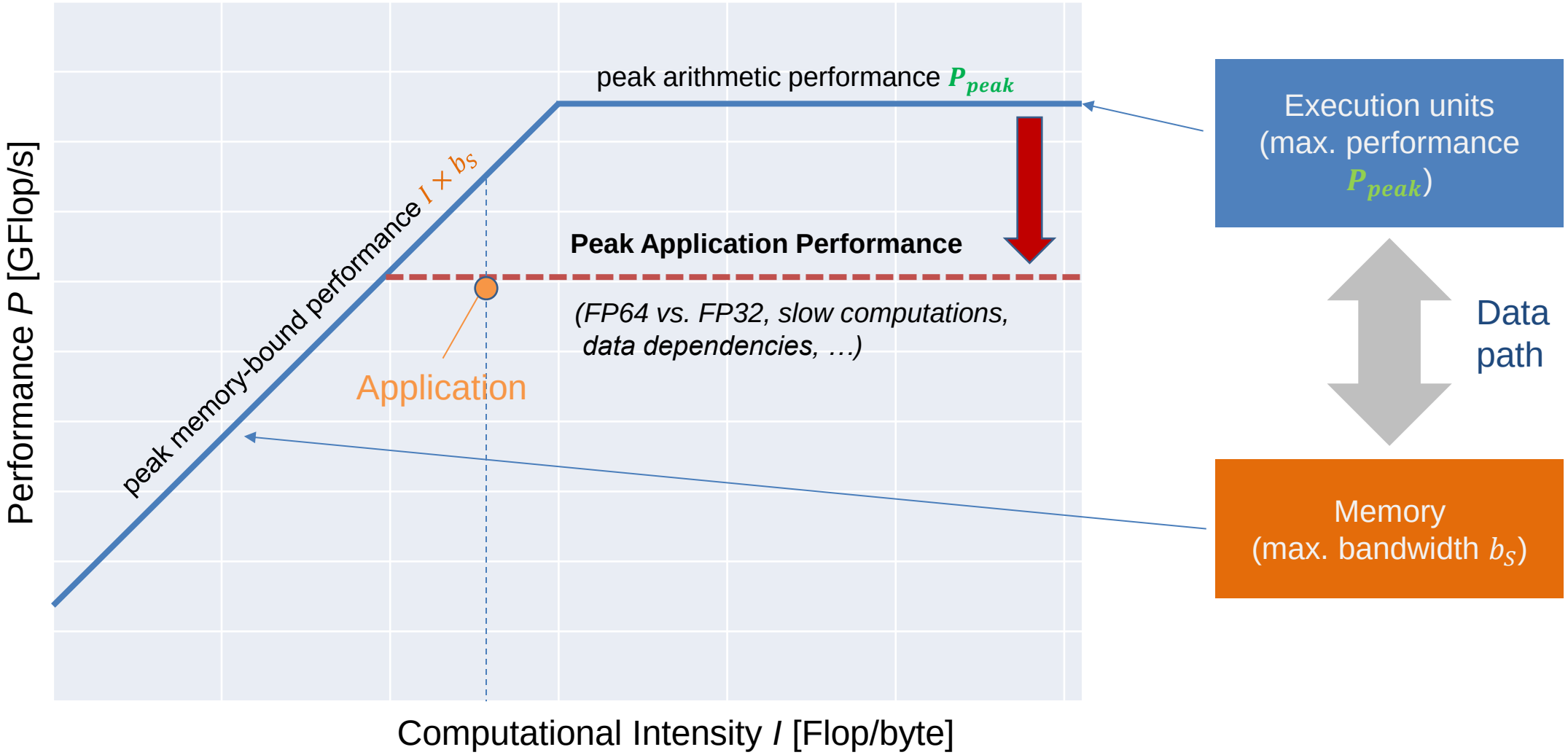
Outline

- Analytical performance modeling
- Basic x86 processor and core architecture
- Code execution on Out-of-order processor cores
- x86 Instruction set intro

- Analysis of simple kernels – demo and hands-on
- Introduction to OSACA
- Arm ISA and A64FX intro
- More complex case studies – demo and hands-on

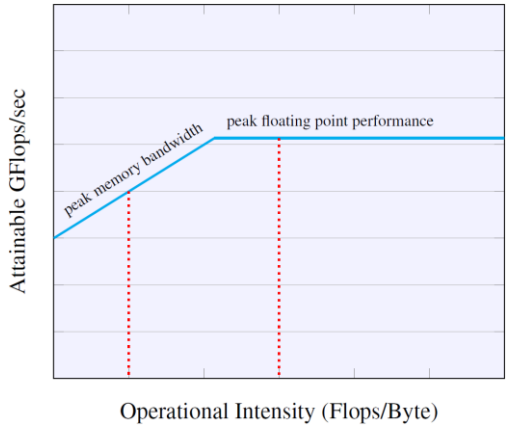
- Summary, caveats, and take-aways

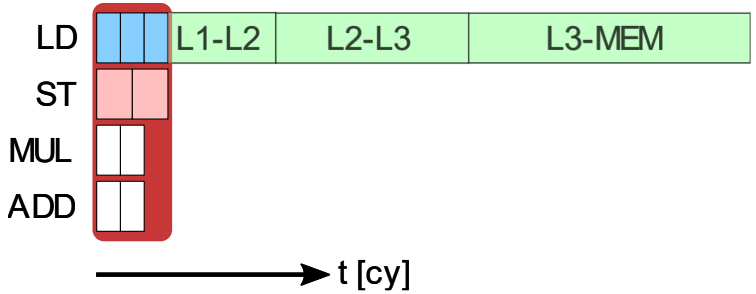
Analytical Performance Modeling



Analytical Performance Modeling

- What is the best performance my code can achieve?
- What are the relevant hardware bottlenecks?
- Apply simplified model of underlying hardware, consisting of
  - in-core execution (covered in this tutorial)
  - data transfer (covered elsewhere)





NHR FAU

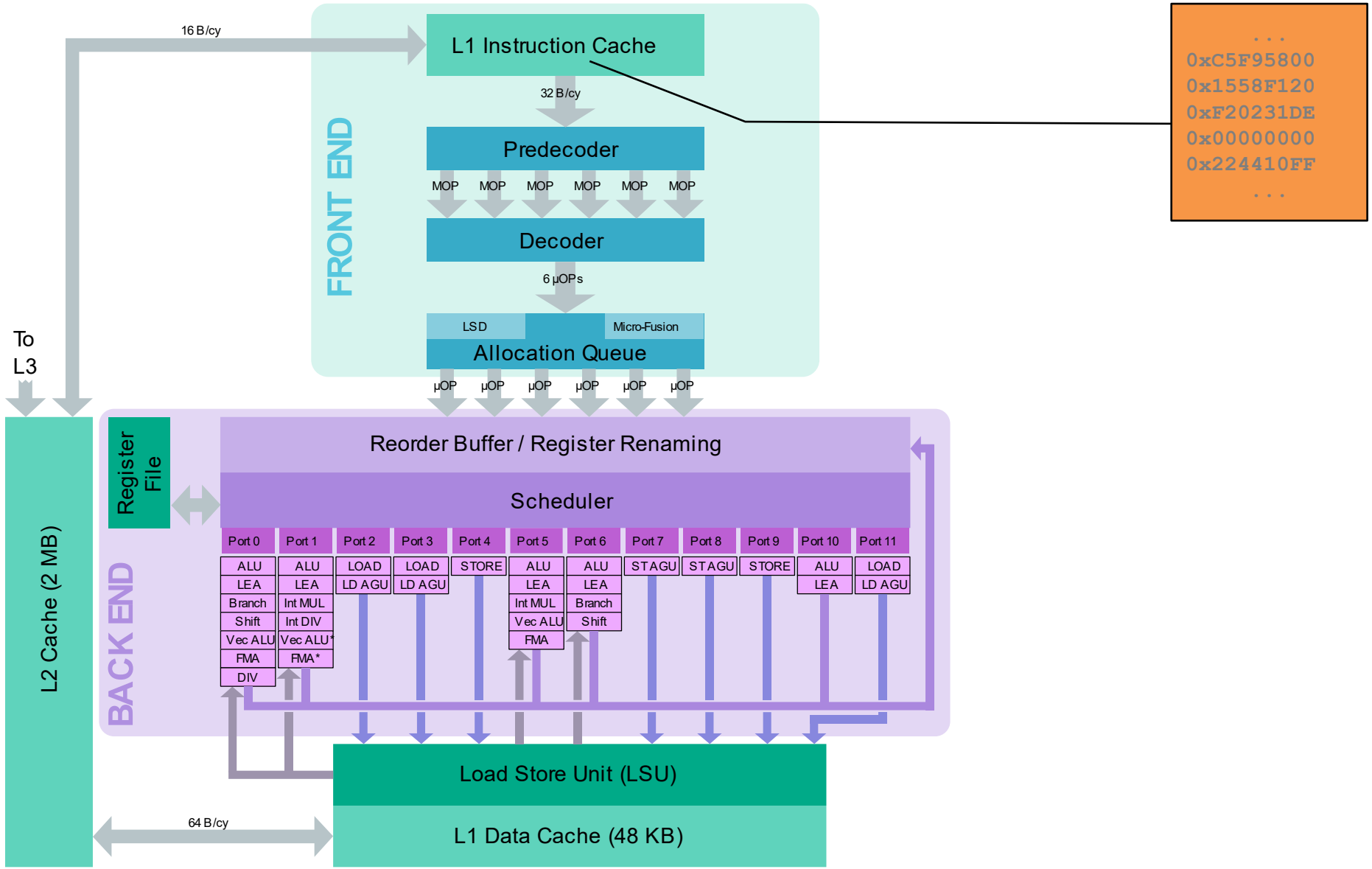
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Basic x86 out-of-order core architecture

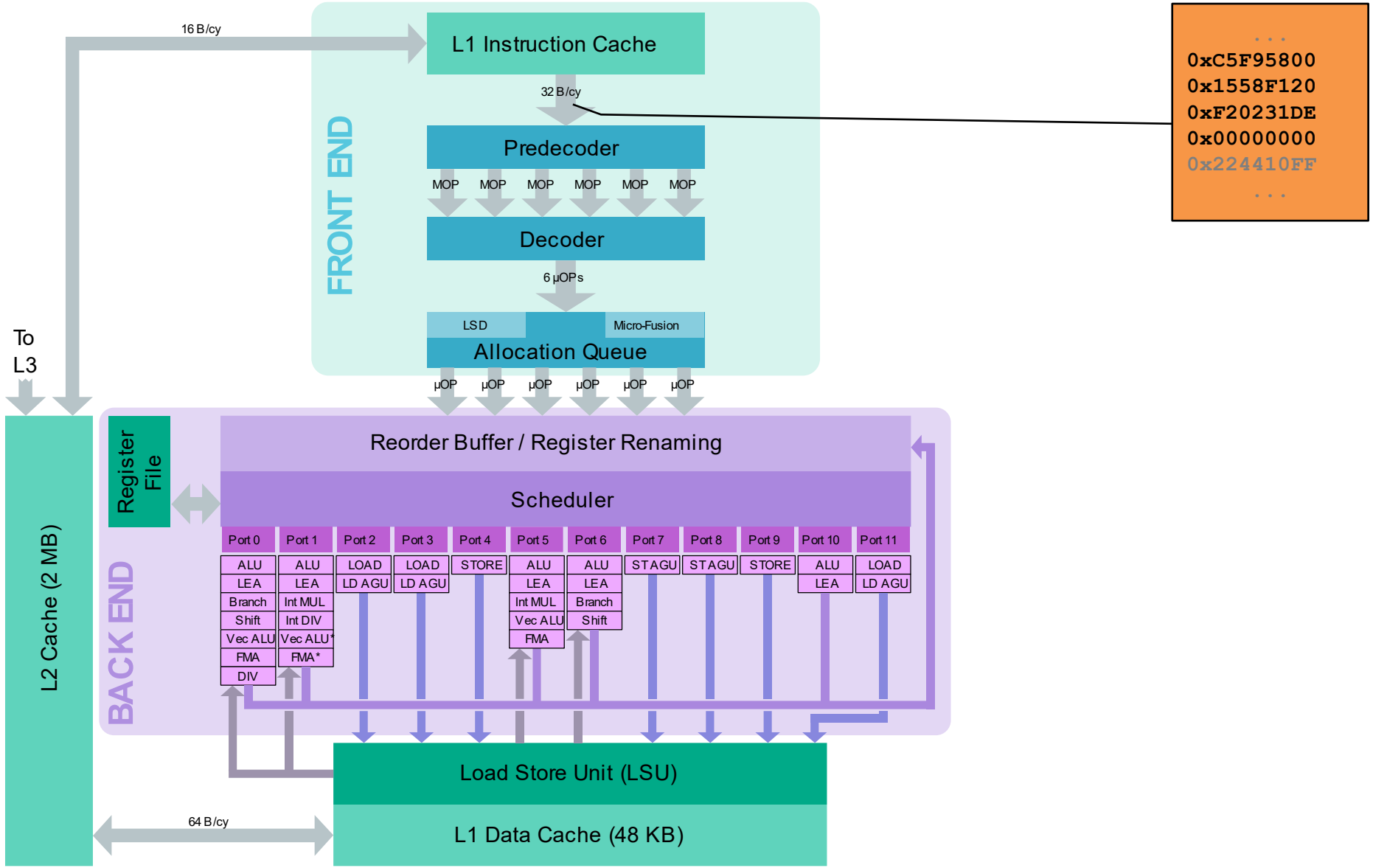
On the example of an Intel Sapphire Rapids (SPR) chip



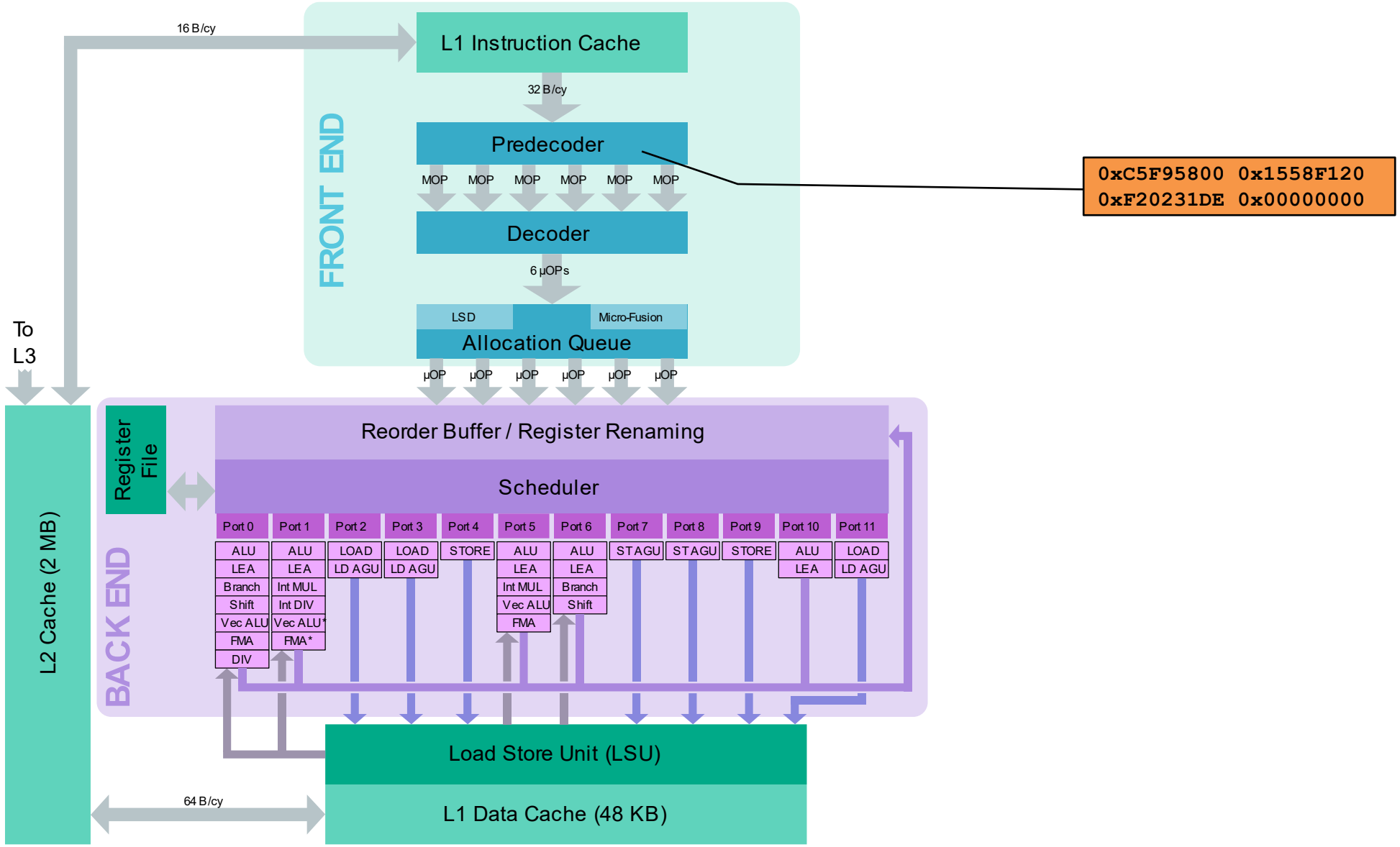
Basic processor and core architecture



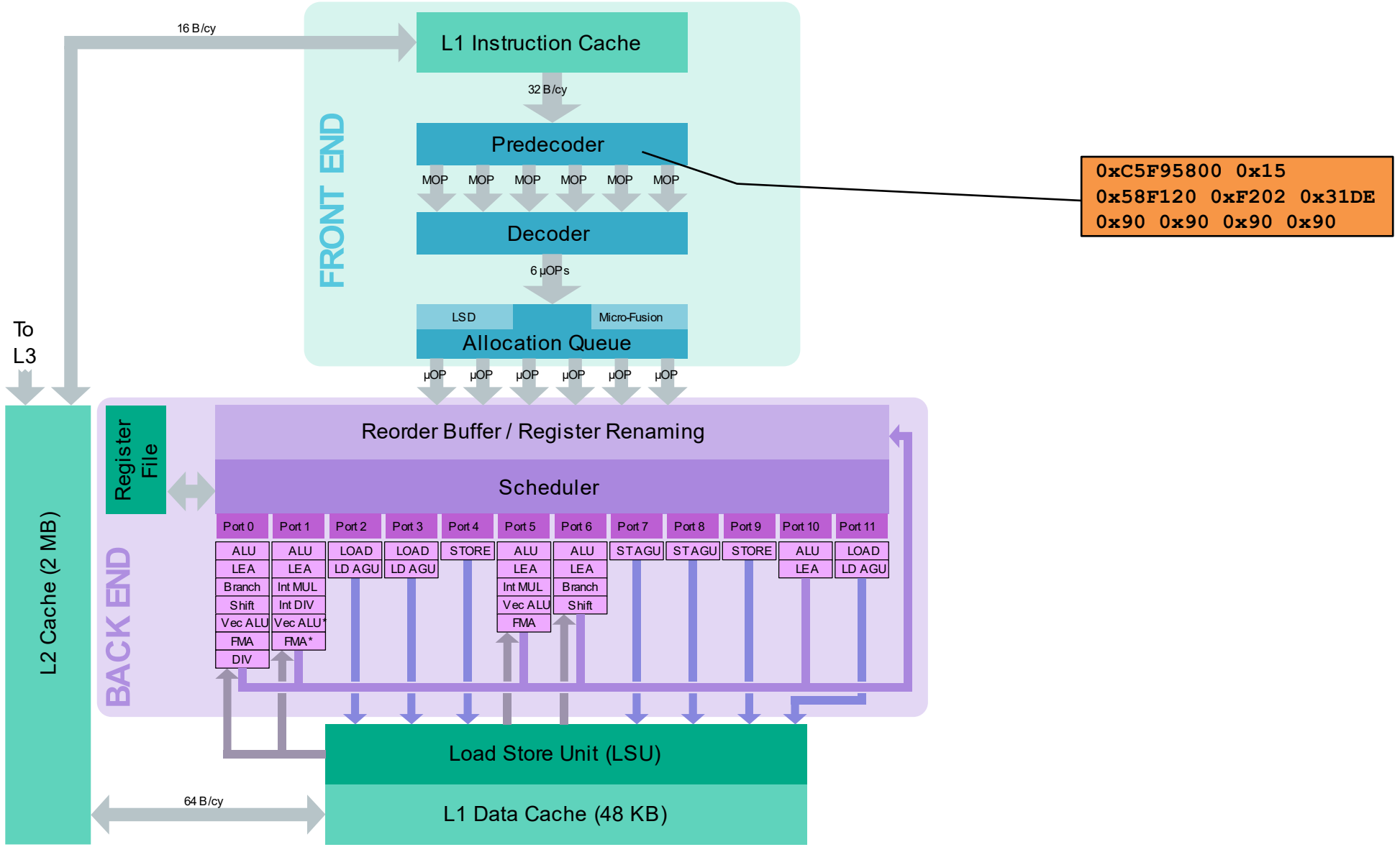
Basic processor and core architecture



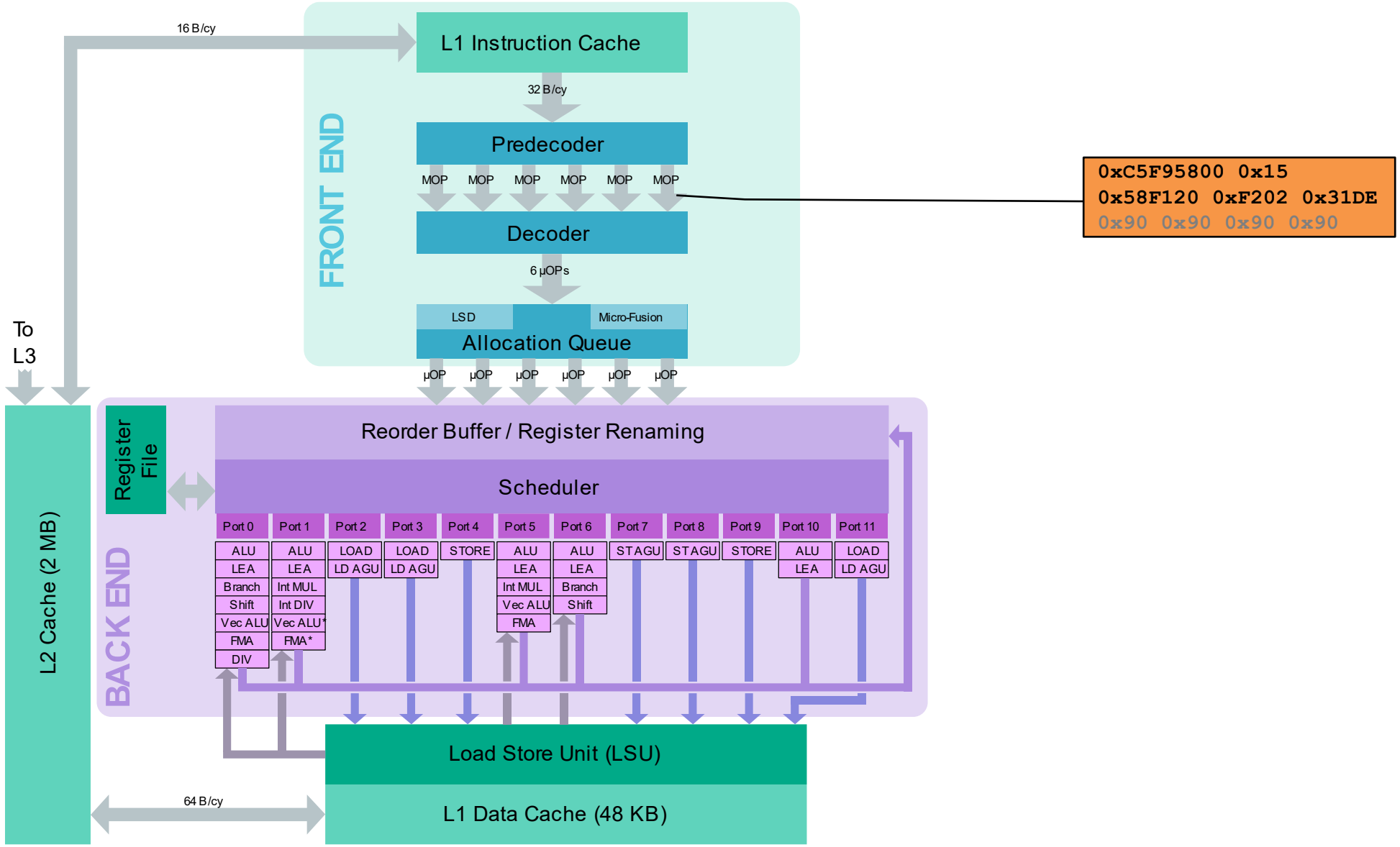
Basic processor and core architecture



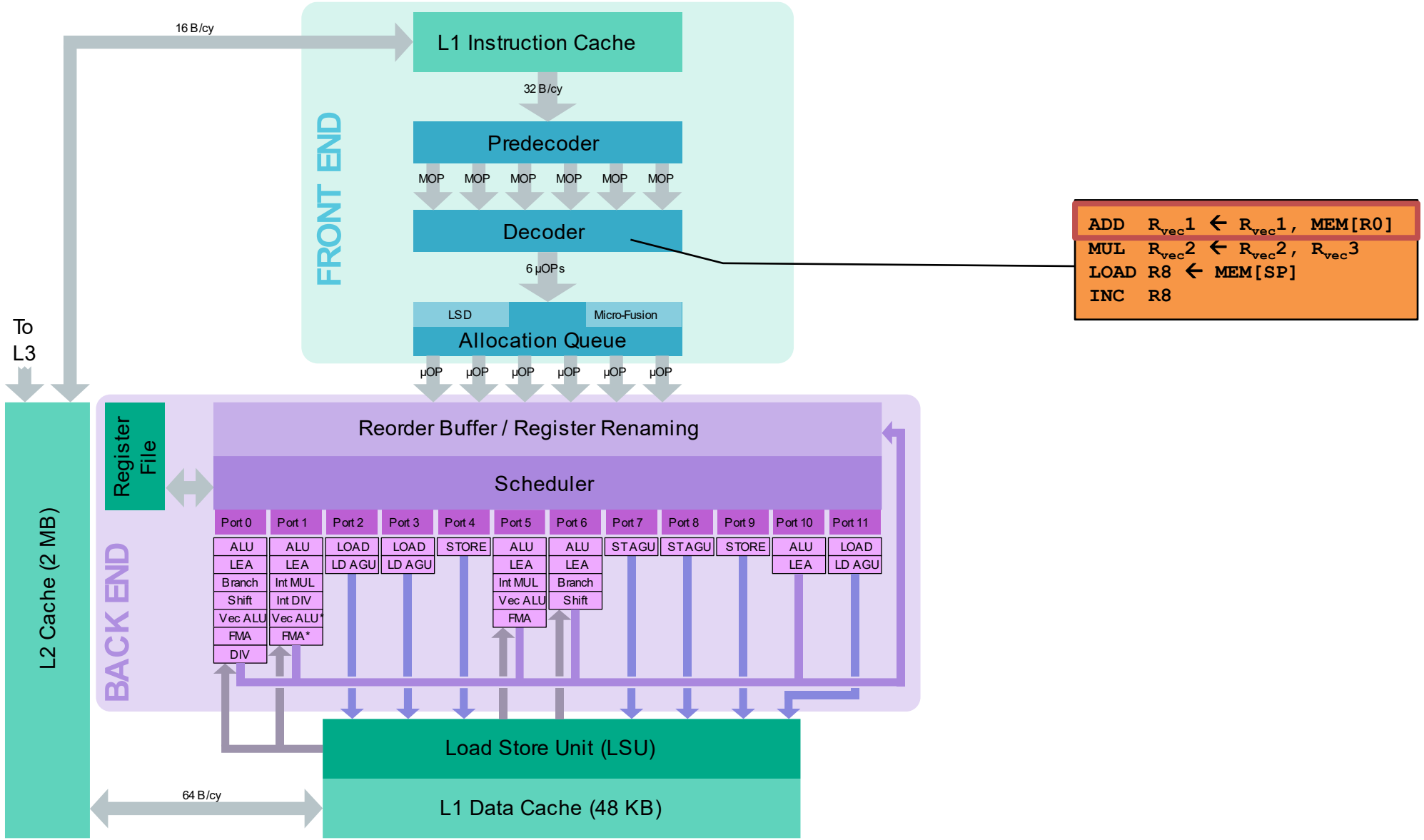
Basic processor and core architecture



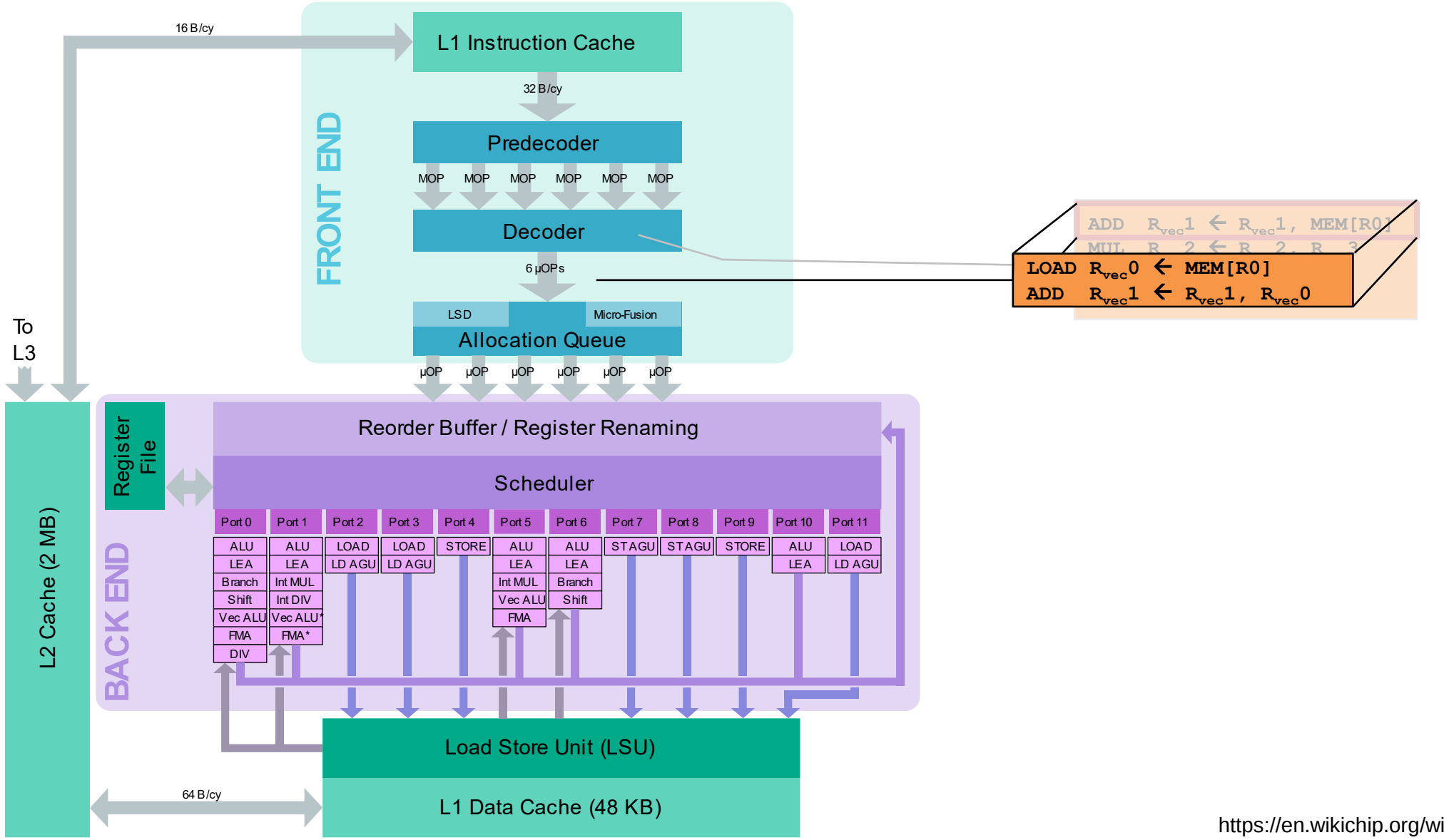
Basic processor and core architecture



Basic processor and core architecture

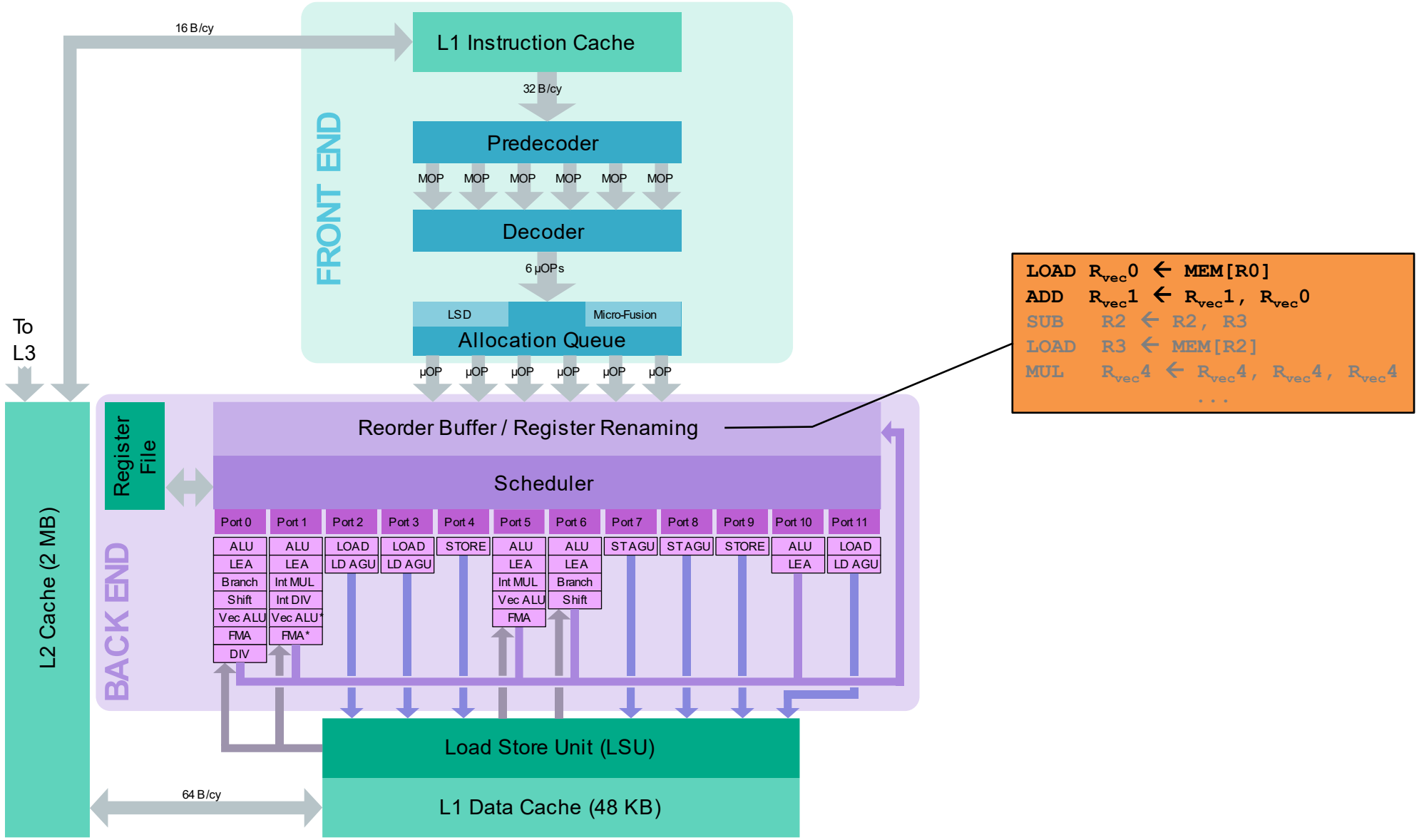


Basic processor and core architecture

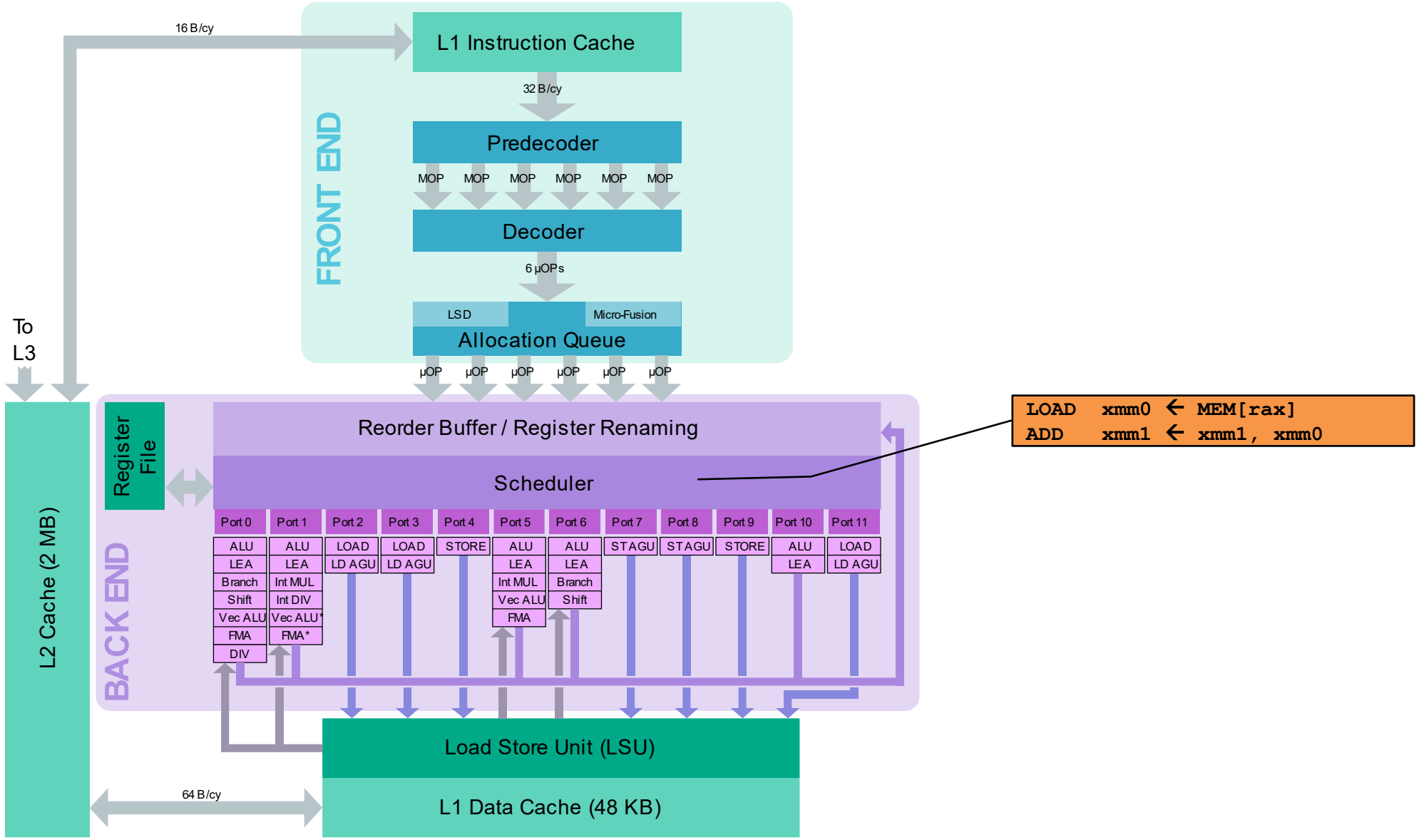


https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

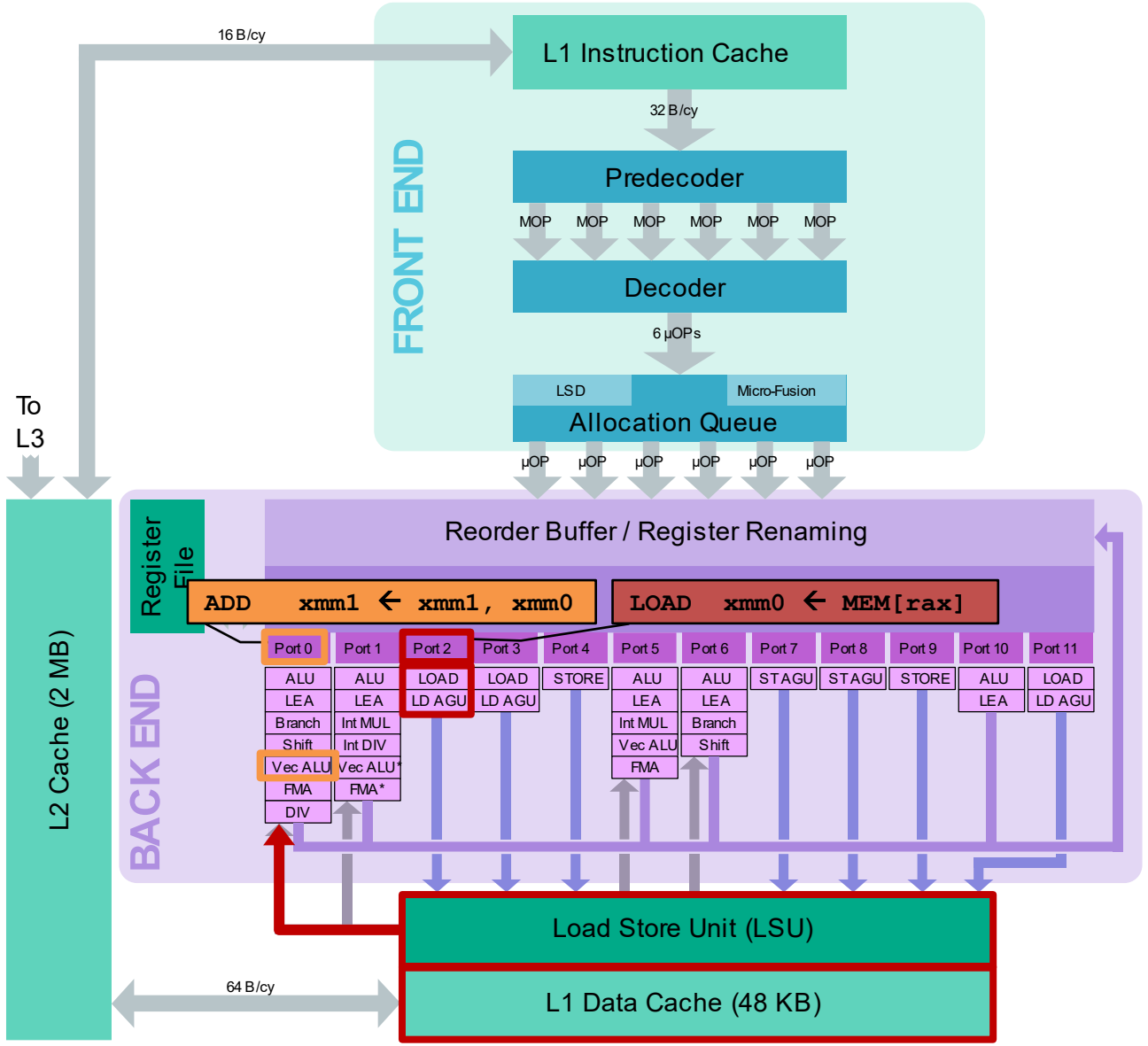
Basic processor and core architecture



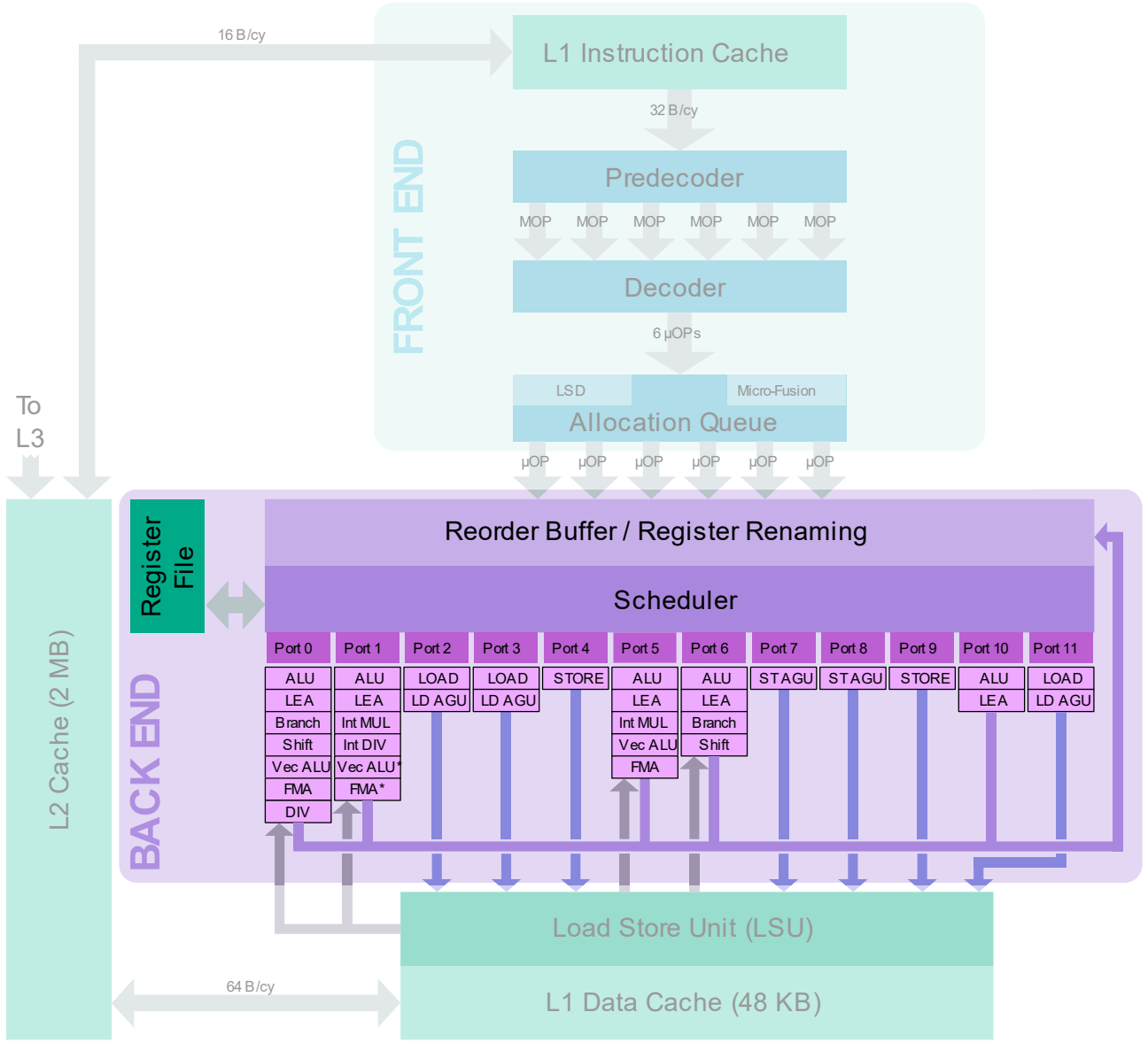
Basic processor and core architecture



Basic processor and core architecture

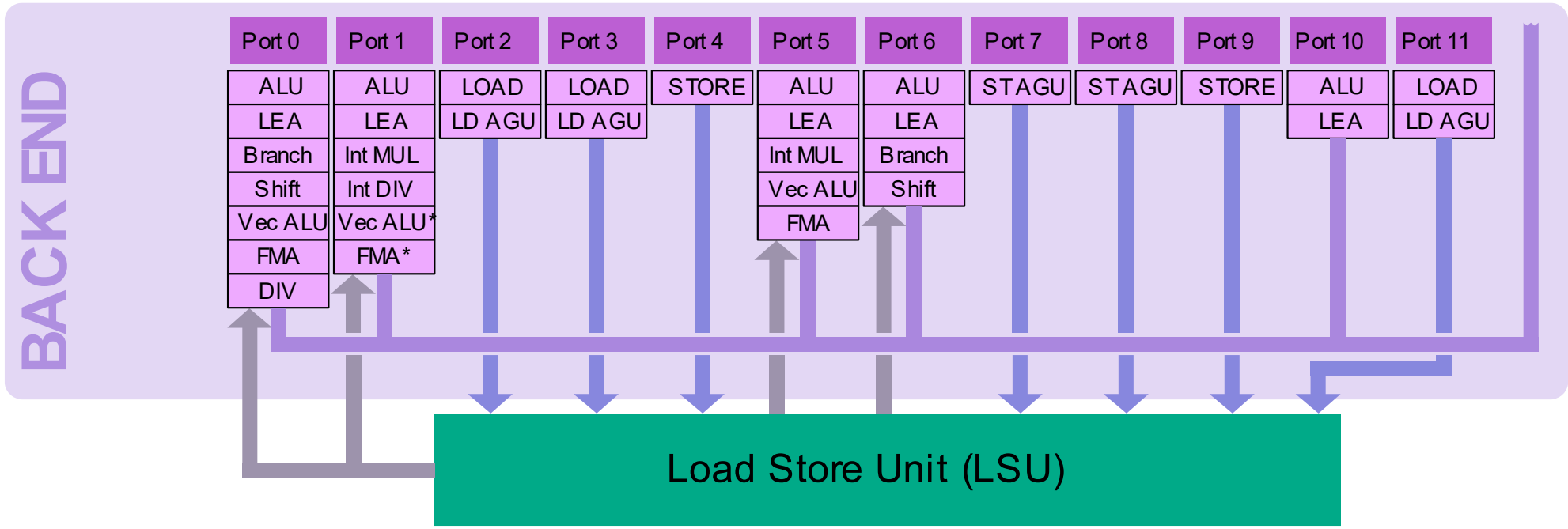


Basic processor and core architecture



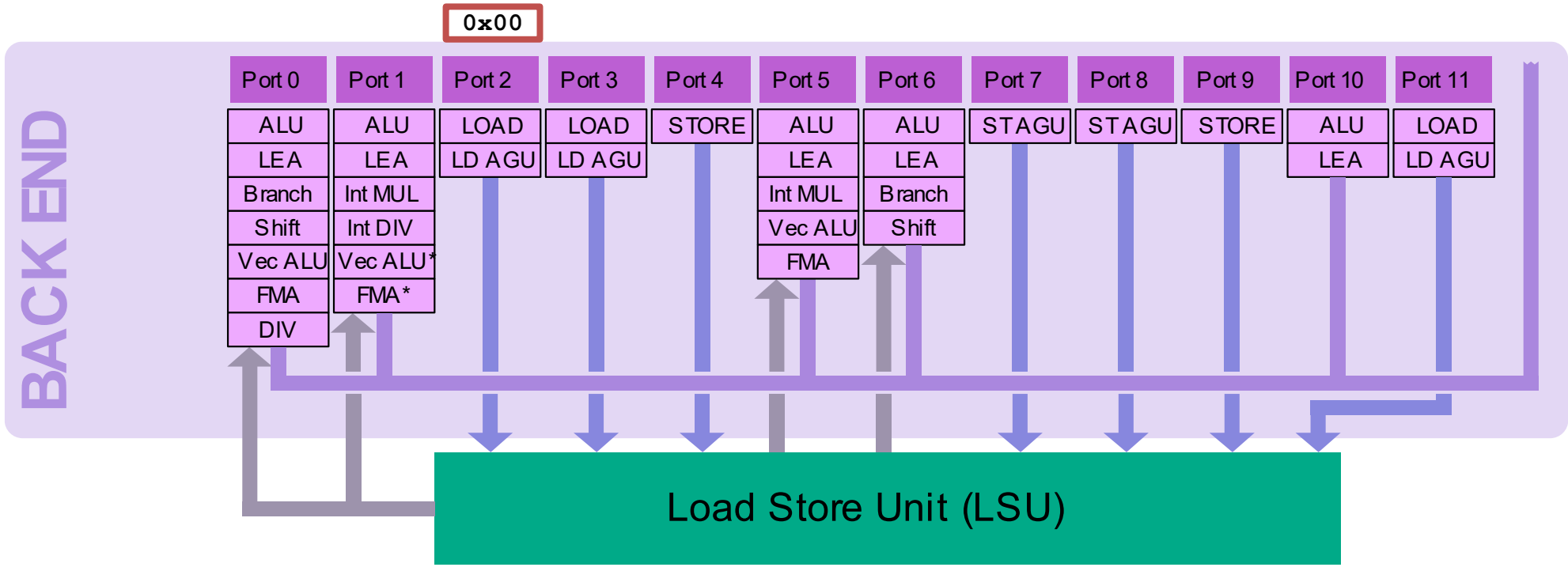
Basic processor and core architecture

```
0x00 LOAD from address 0x1f8223de to reg1          1cy on 2|3|11
0x04 LOAD from address 0x1f8244de to reg2          1cy on 2|3|11
0x08 ADD reg1 and reg2 and save in reg3          1cy on 0|1|5|6|10
0x0C STORE reg3 to address 0x2010ff08        1cy on 4|9, 1cy on 7|8
```



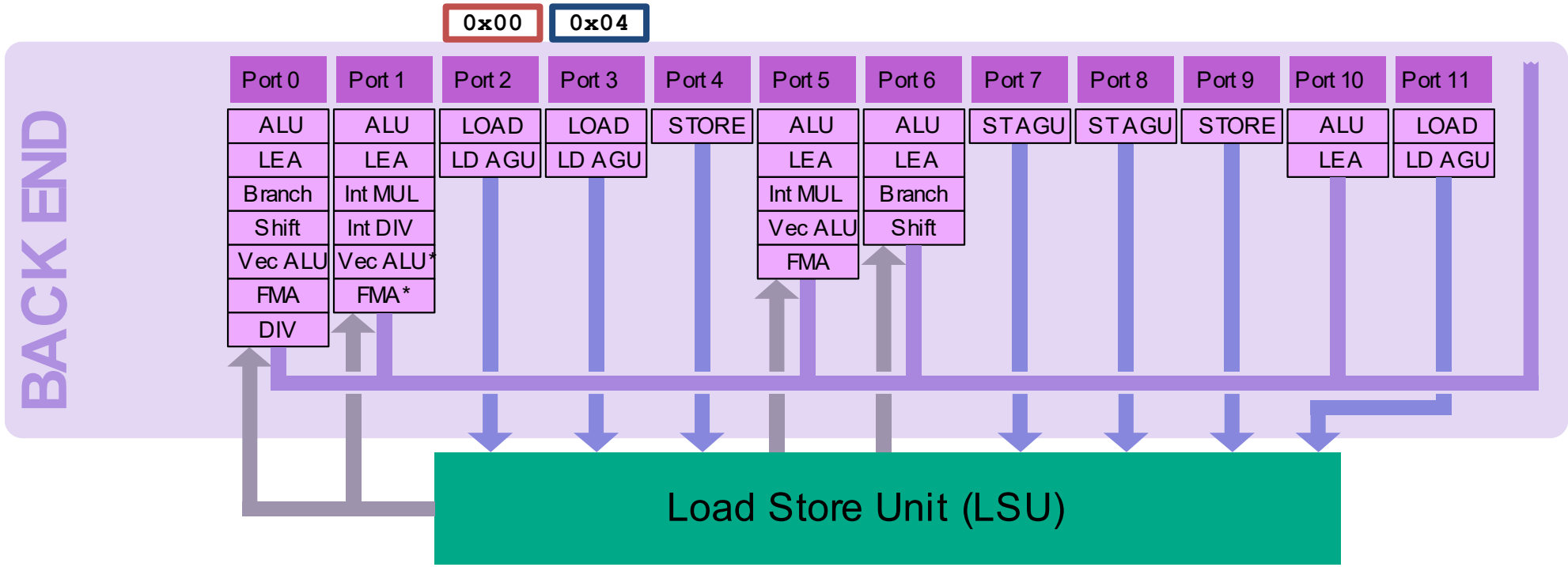
Basic processor and core architecture

```
0x00 LOAD from address 0x1f8223de to reg1                1cy on 2|3|11
0x04 LOAD from address 0x1f8244de to reg2                1cy on 2|3|11
0x08 ADD reg1 and reg2 and save in reg3                  1cy on 0|1|5|6|10
0x0C STORE reg3 to address 0x2010ff08                    1cy on 4|9, 1cy on 7|8
```



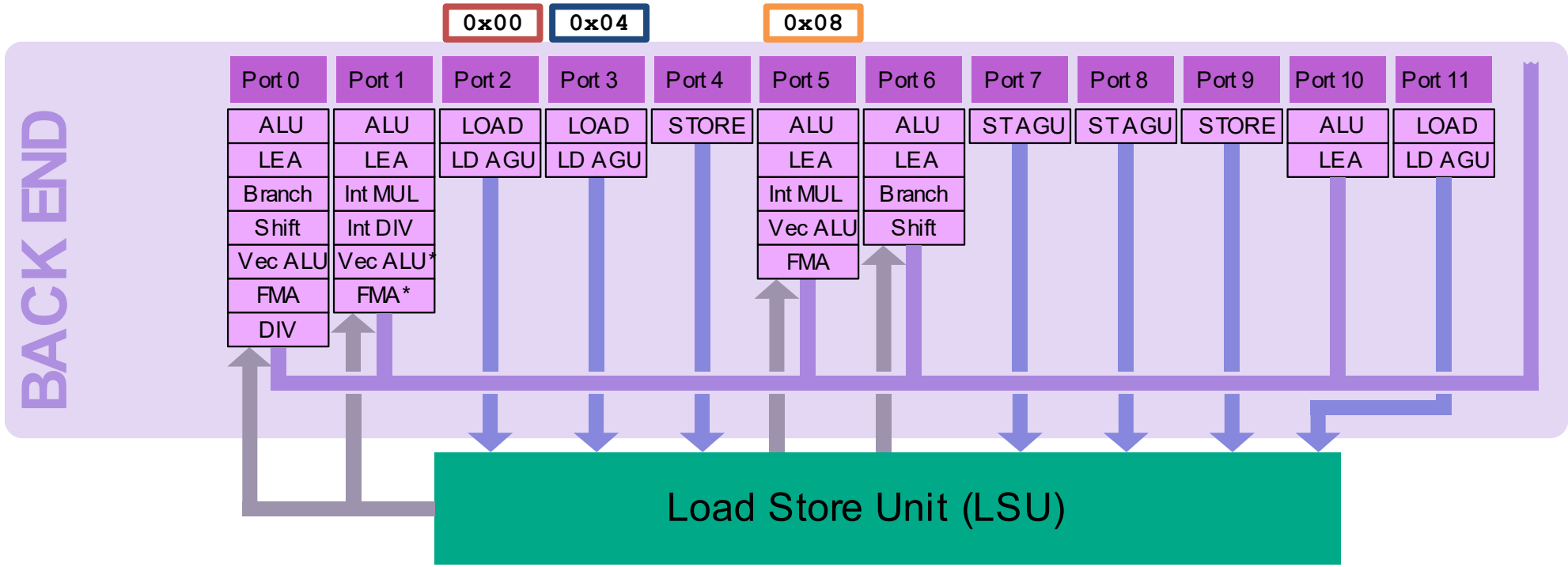
Basic processor and core architecture

```
0x00 LOAD from address 0x1f8223de to reg1                1cy on 2|3|11
0x04 LOAD from address 0x1f8244de to reg2                1cy on 2|3|11
0x08 ADD reg1 and reg2 and save in reg3                  1cy on 0|1|5|6|10
0x0C STORE reg3 to address 0x2010ff08                    1cy on 4|9, 1cy on 7|8
```



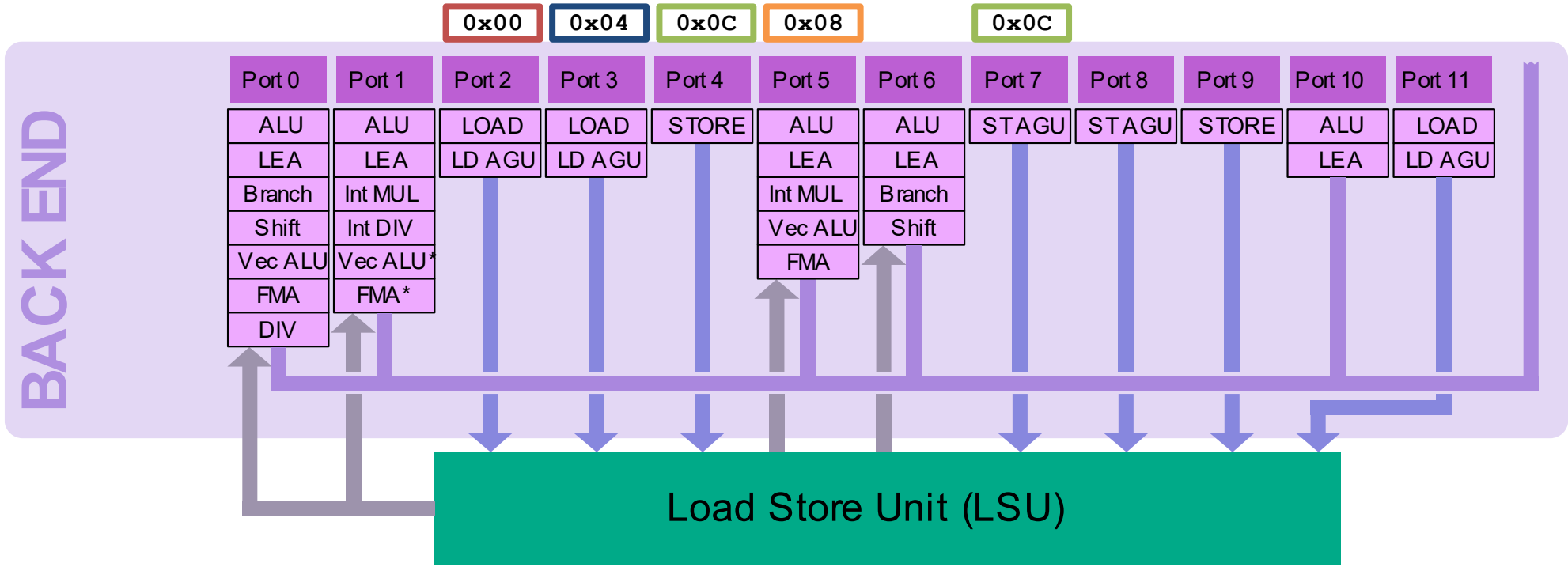
Basic processor and core architecture

```
0x00 LOAD from address 0x1f8223de to reg1          1cy on 2|3|11
0x04 LOAD from address 0x1f8244de to reg2          1cy on 2|3|11
0x08 ADD reg1 and reg2 and save in reg3          1cy on 0|1|5|6|10
0x0C STORE reg3 to address 0x2010ff08      1cy on 4|9, 1cy on 7|8
```



Basic processor and core architecture

```
0x00 LOAD from address 0x1f8223de to reg1          1cy on 2|3|11
0x04 LOAD from address 0x1f8244de to reg2          1cy on 2|3|11
0x08 ADD reg1 and reg2 and save in reg3          1cy on 0|1|5|6|10
0x0C STORE reg3 to address 0x2010ff08          1cy on 4|9, 1cy on 7|8
```



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Code execution on out-of-order CPUs

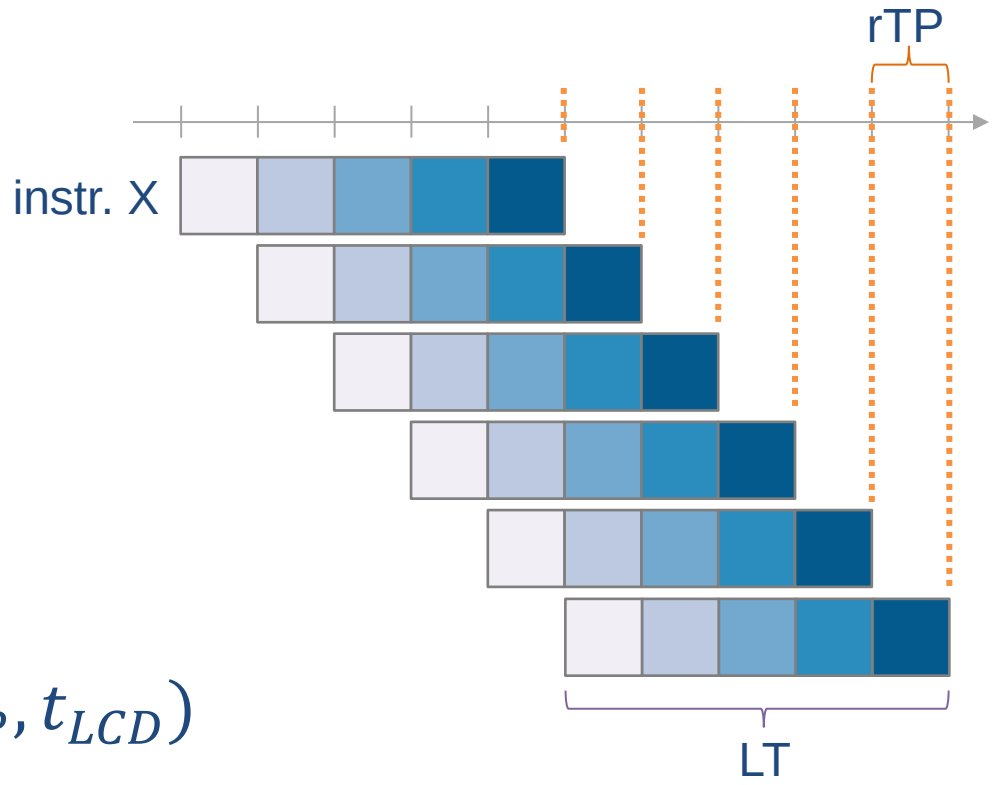
Terminology and explanation



Code execution on OoO processor cores

Three metrics to estimate the in-core runtime:

- **Reciprocal Throughput** (rTP)
- **Latency** (LT) and **Critical Path** (CP)
- **Loop-carried dependencies** (LCD)



Simplified runtime estimation: $t_c = \max(t_{rTP}, t_{LCD})$

Unit "**it**": 1 high-level iteration

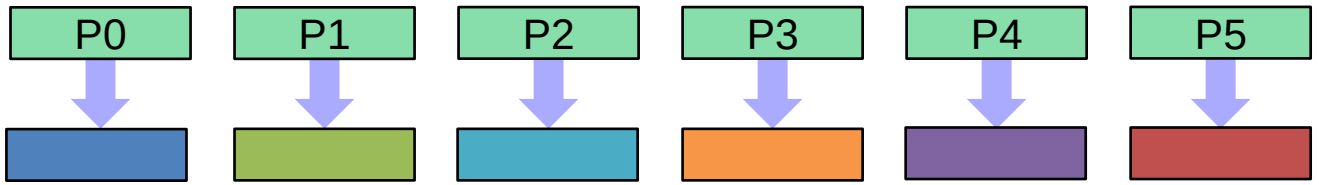
One assembly loop can easily consist of dozens of high-level iterations, e.g.:
8x vectorized, 4x unrolled
→ 1 assembly iteration = 32 it

Code execution on OoO processor cores

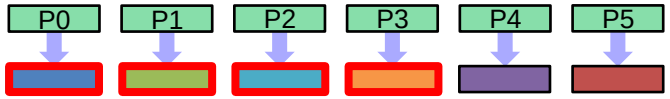
- Simple HW model:
  - Six types of functional units (i.e., types of instructions), each functional unit (FU) assigned to one port:



  - Reciprocal throughput for each instruction: 1cy
  - Latency for each instruction: 1cy
  - Port model:

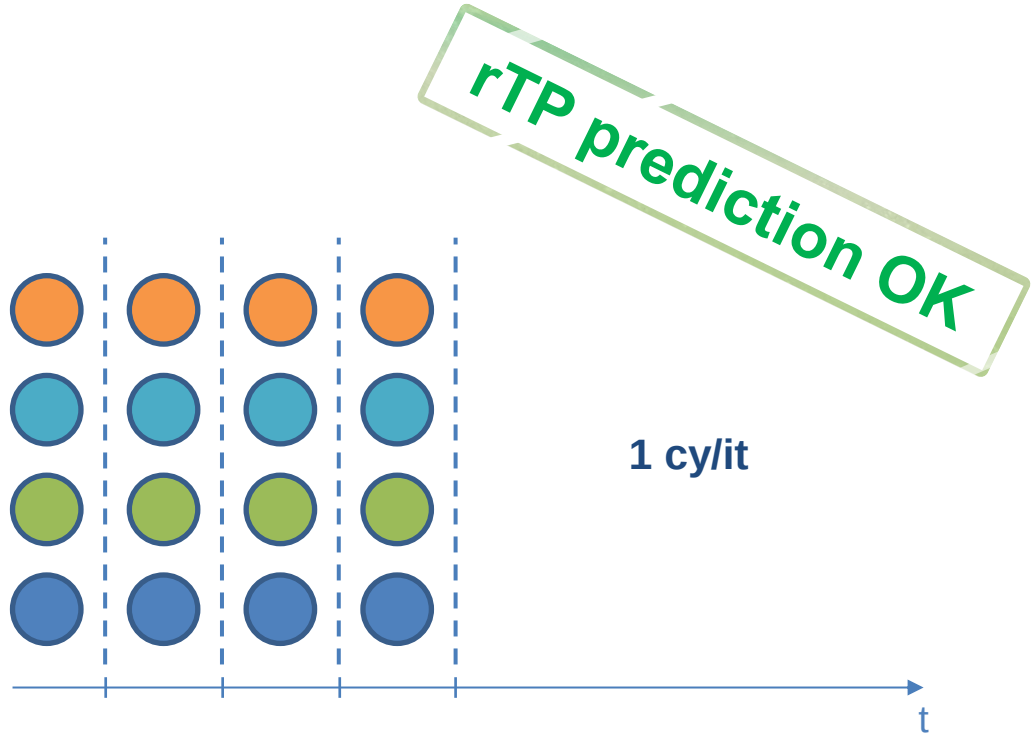


Code execution on OoO processor cores



- Loop 1: 
  - No dependencies within loop
  - No intra-loop dependencies

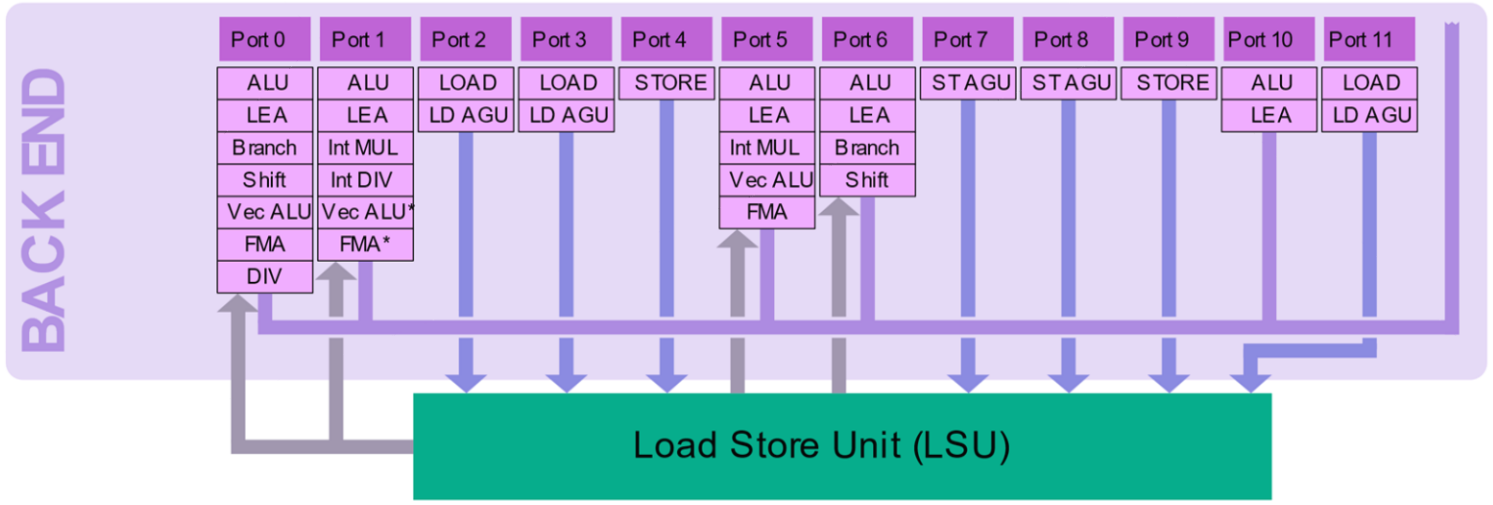
  - rTP prediction: 1 cy
  - CP prediction: 1 cy
  - LCD prediction: -



Remember slides 17-21?

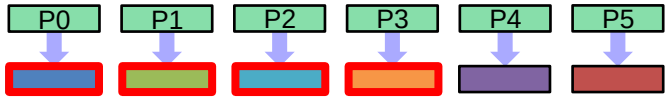
Basic processor and core architecture

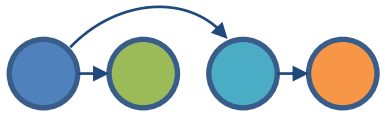
```
0x00 LOAD from address 0x1f8223de to reg1          1cy on 2|3|11
0x04 LOAD from address 0x1f8244de to reg2          1cy on 2|3|11
0x08 ADD reg1 and reg2 and save it reg3          1cy on 0|1|5|6|10
0x0C STORE reg3 to address 0x2010ff08     1cy on 4|9, 1cy on 7|8
```



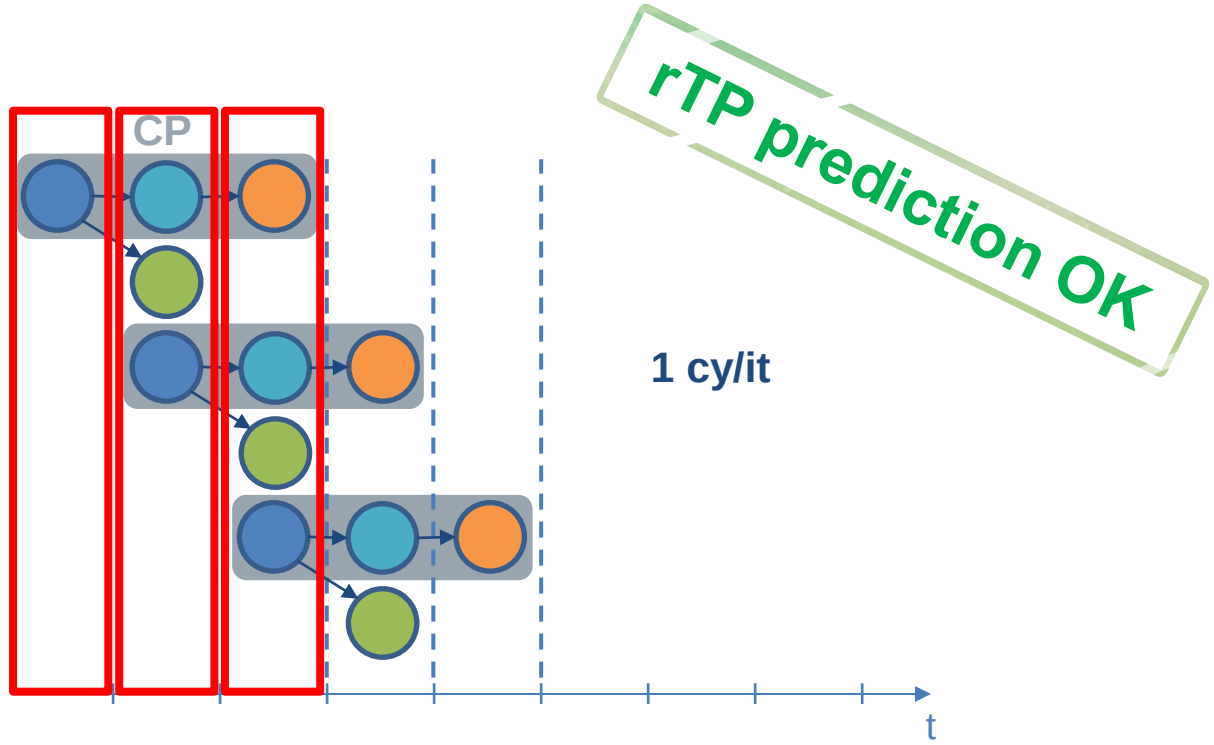
Core-Level Performance Engineering Tutorial | NHR@FAU 17

Code execution on OoO processor cores

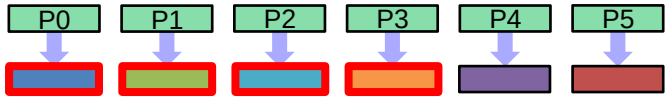


- Loop 2: 
  - Dependencies within loop body
  - No loop-carried dependencies

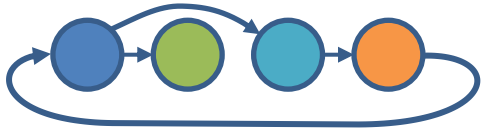
  - rTP prediction: 1 cy
  - CP prediction: 3 cy
  - LCD prediction: -



Code execution on OoO processor cores



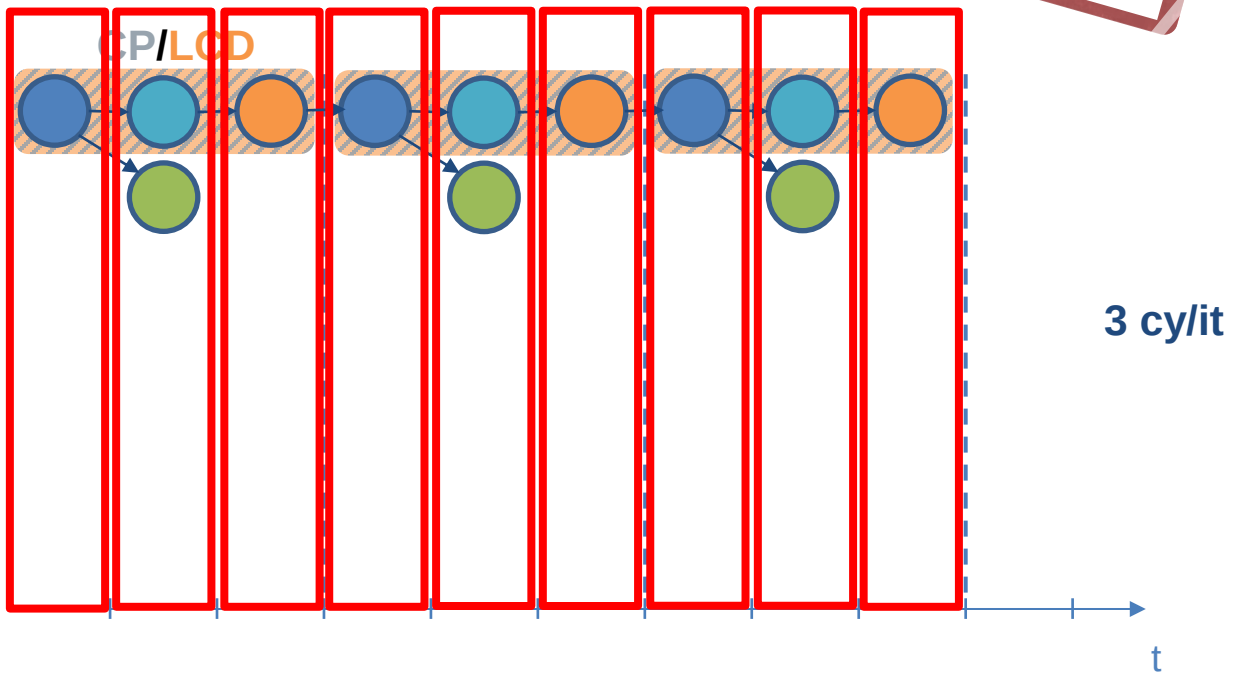
- Loop 3:



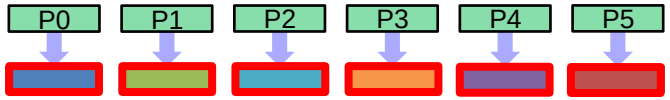
  - Dependencies within loop body
  - Loop-carried dependency

  - rTP prediction: 1 cy
  - CP prediction: 3 cy
  - LCD prediction: 3 cy

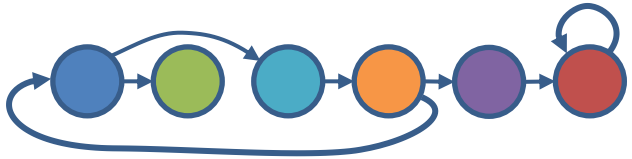
rTP prediction not sufficient



Code execution on OoO processor cores



- Loop 4:

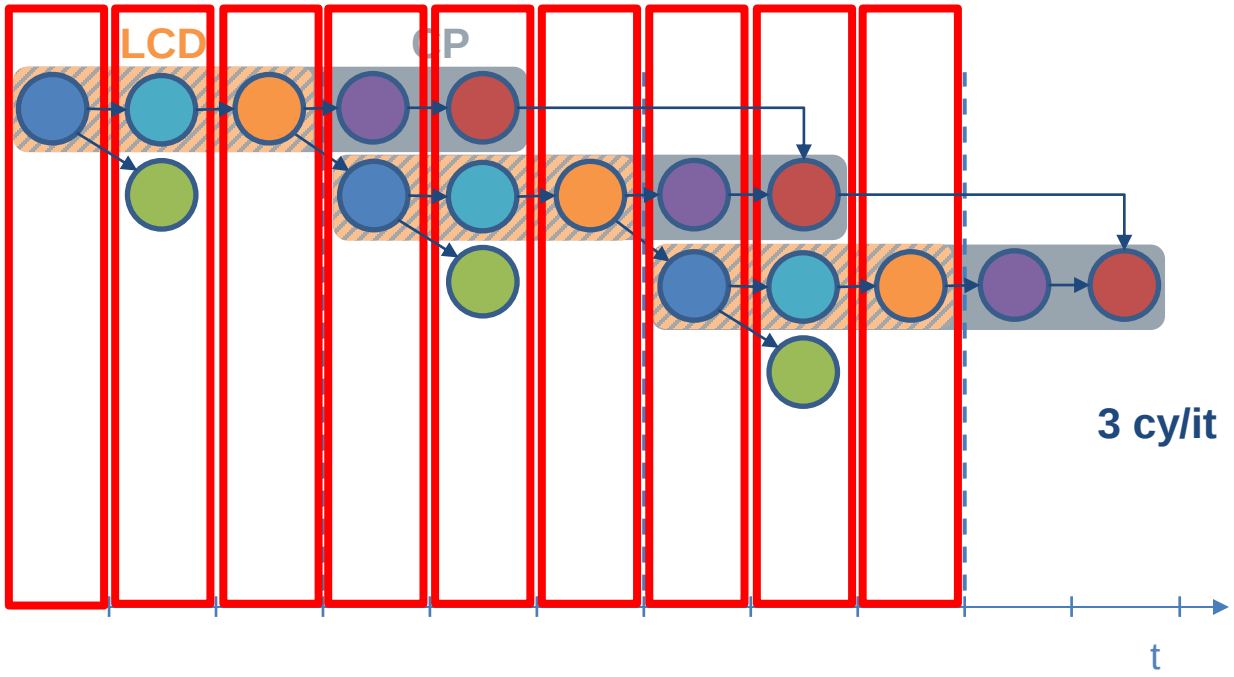


  - Dependencies within loop body
  - Loop-carried dependency

  - rTP prediction: 1 cy
  - CP prediction: 5 cy
  - LCD prediction: 3 cy

≠

  - Other limitations:
    - Reorder buffer
    - Loop length
    - Resources (not enough ports, …)
    - Decoder
    - Data
    - …



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Hands-On #0: Out-of-Order Execution

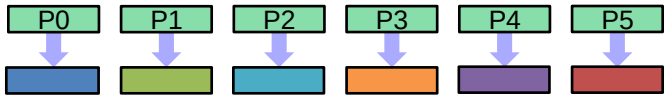
➔ https://go-nhr.de/CLPE-ex0



Hands-On: Out-of-Order Execution

Dot product

Machine model:



Instructions:

each with a reciprocal throughput and latency of 1 cy

→ **Moodle, hands-on #0 (both Multiple-Choice and Drag&Drop)**

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Break



NHR FAU

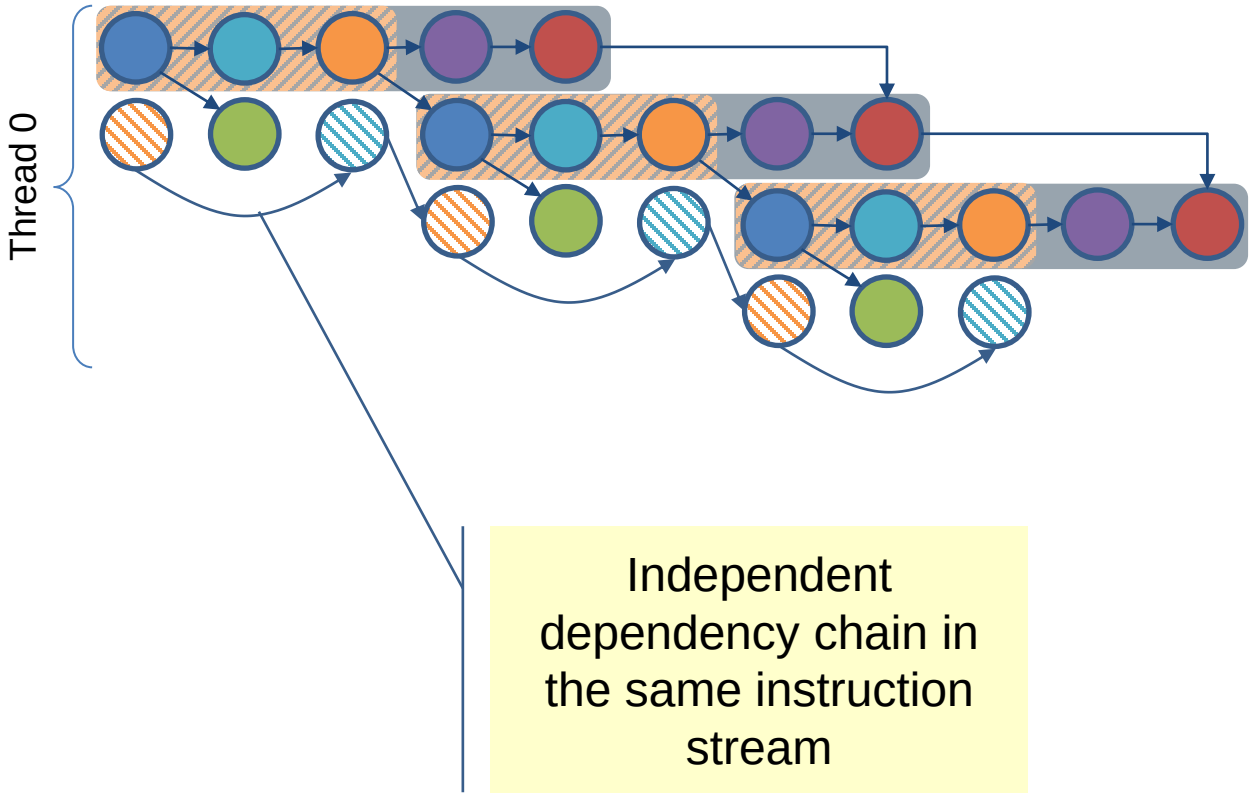
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Simultaneous Multi-Threading (SMT)

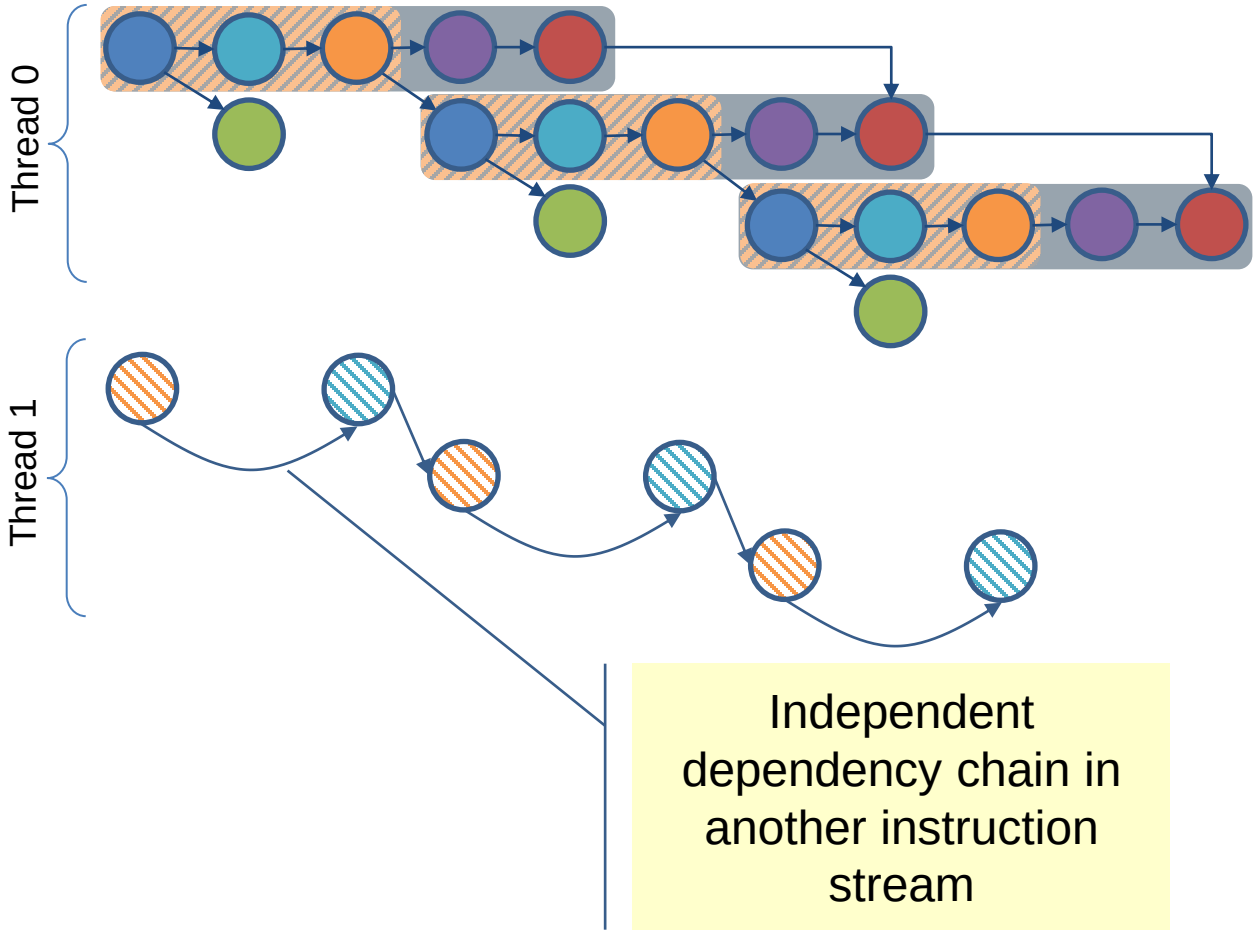


Two ways to fill pipeline bubbles

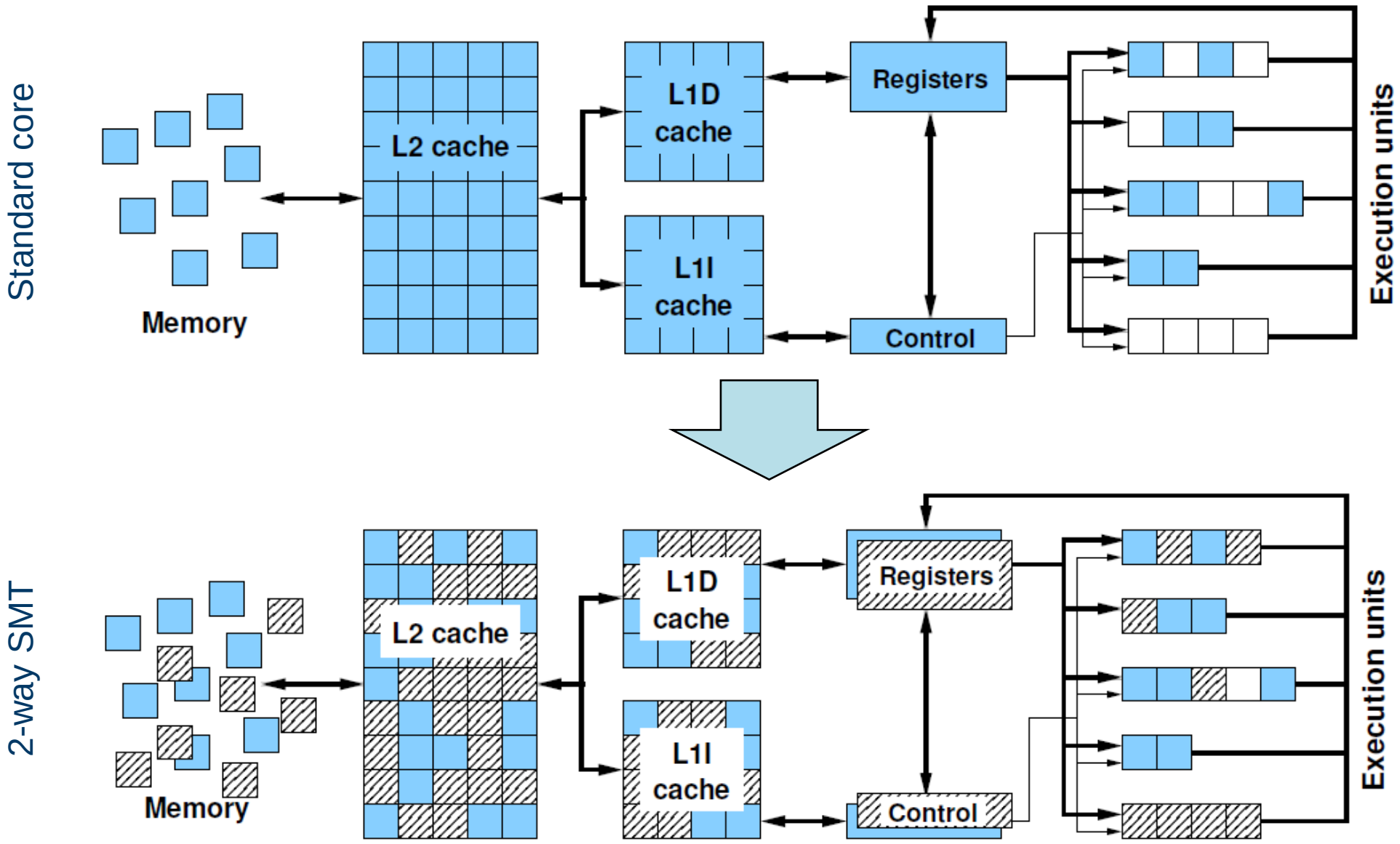
Put more independent instructions in the loop kernel:



Run several independent threads on the same core:



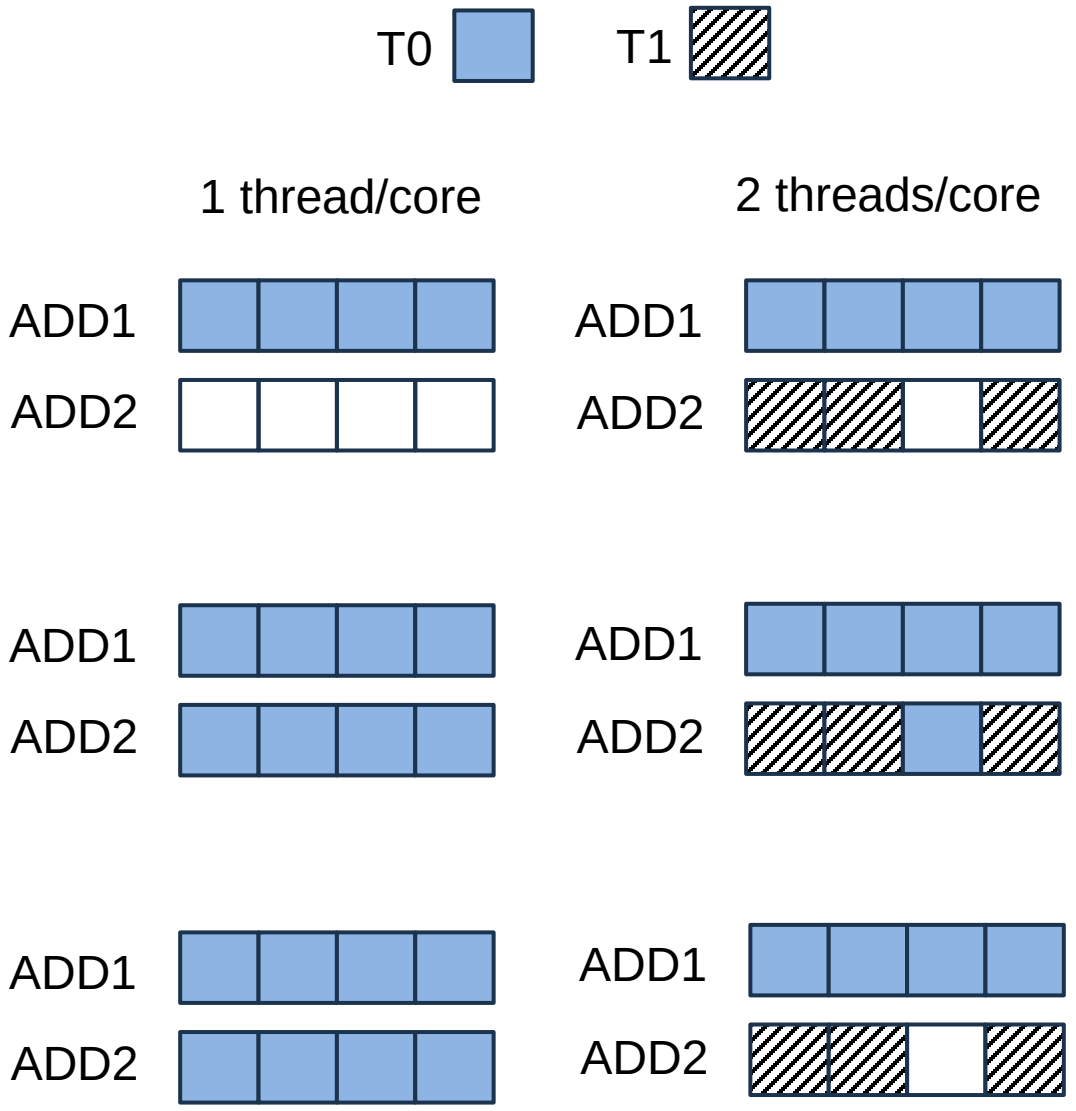
Simultaneous Multi-Threading (SMT) basic principle



SMT: the details

How are core resources split among software threads? Example: two ADD units

- Static: Each thread gets half the pipelines, buffers, queues, etc. as soon as SMT is activated (at boot)
- Fully dynamic: Resources are split dynamically between threads cycle-by-cycle (current x86)
- On-demand static: (Some) resources are split statically as soon as more than one thread is running (NVIDIA Vera)



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Introduction to the x86 ISA (Instruction Set Architecture)



Basics of the x86-64 ISA

- Instructions have 0 to 5 operands (and possibly more suppressed operands)
- Operands can be registers (%), memory references ((...)) or immediates ($)
- Opcodes (binary representation of instructions) vary from 1 to 15 bytes

- There are two assembler syntax forms: Intel (left) and AT&T (right)
- Addressing Mode:
  - Intel: BASE + INDEX * SCALE + DISPLACEMENT
  - AT&T: DISPLACEMENT(BASE, INDEX, SCALE)
- C: **`A[i]`** equivalent to **`*(A+i)`** (a pointer has a type: **`A+i*8`**)
- Suffixes: AT&T often uses (optional) suffixes based on the operand size
  - **`b`** (byte): 8 bits, **`w`** (word): 16 bits, **`l`** (long): 32 bits, **`q`** (quad): 64 bits

Intel syntax

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

AT&T syntax

```
movaps    %xmm3, 48(%rdi,%rax,8)
addq      $8, %rax
js        ..B1.4
```

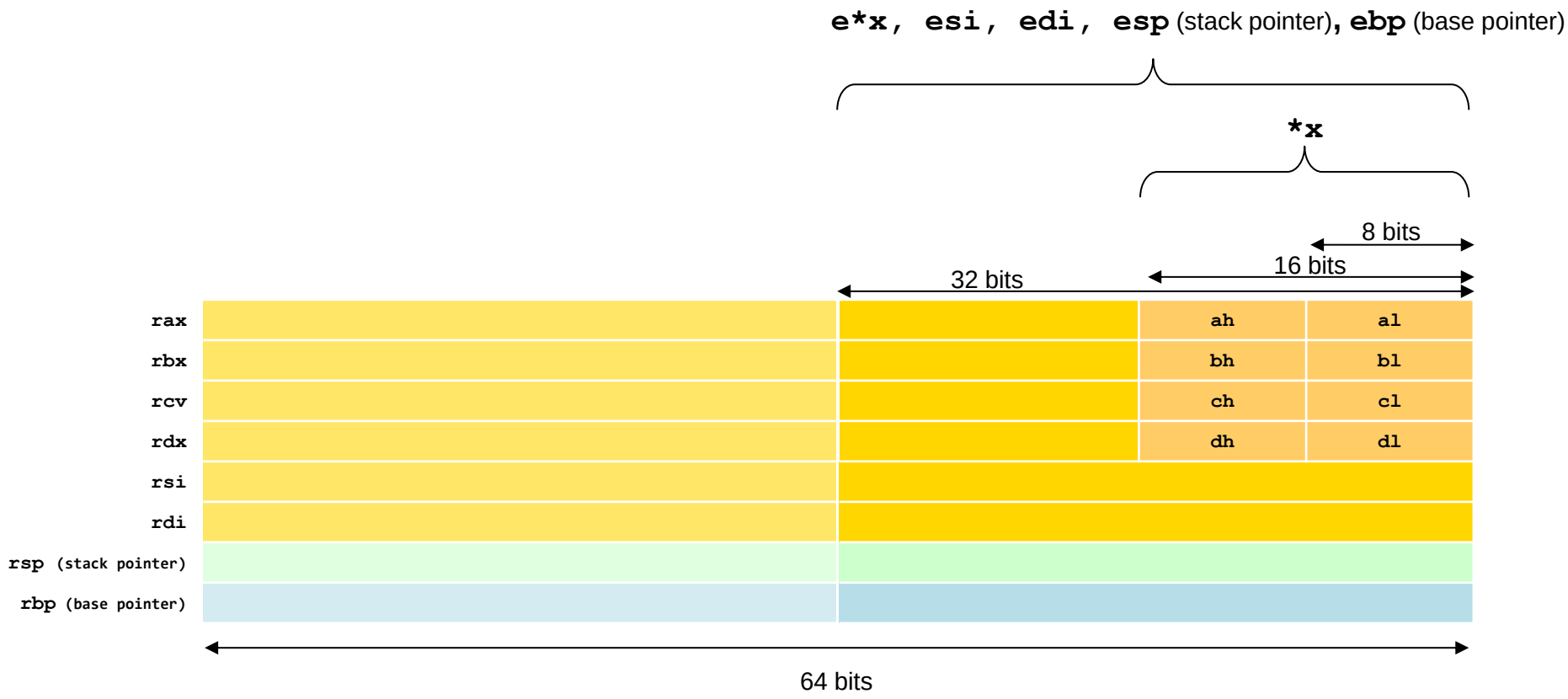
Basics of the x86-64 ISA with extensions

16 general purpose registers (64bit):

rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15

alias with eight 32-bit register set:

eax, ebx, ecx, edx, esi, edi, esp, ebp



Basics of the x86-64 ISA with extensions

Floating Point SIMD registers (aliased):

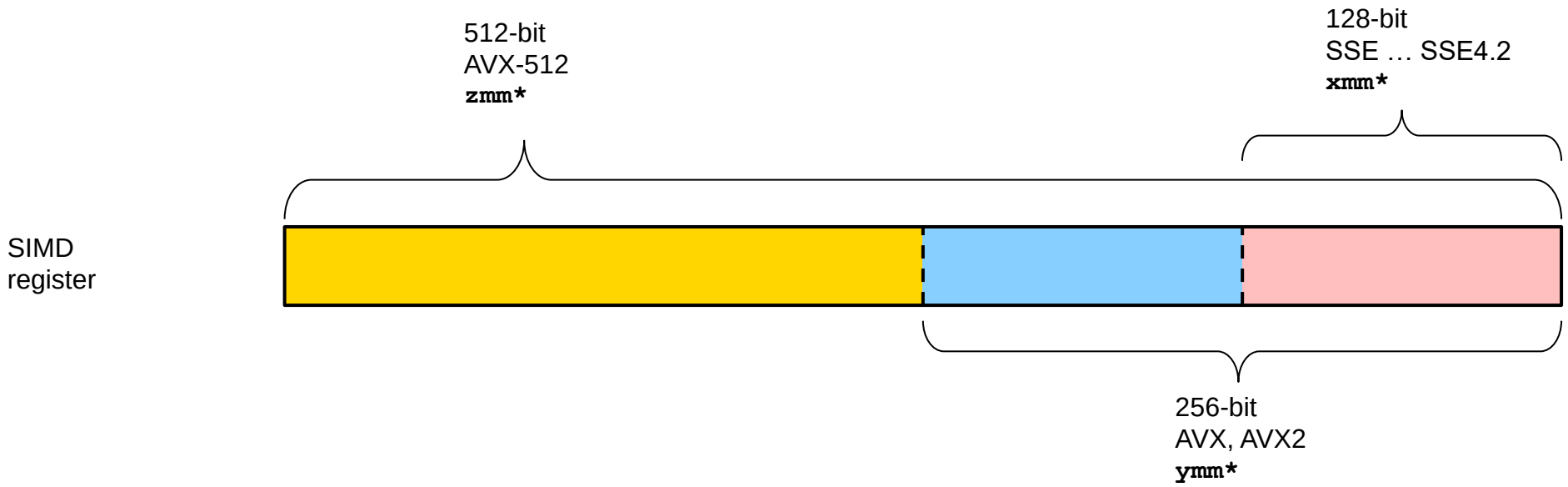
`xmm0-xmm15 (...xmm31)` SSE (128bit)

`ymm0-ymm15 (...ymm31)` AVX (256bit)

`zmm0-zmm31` AVX-512 (512bit)

8 opmask registers (64 bit, AVX512 only):

`k0-k7`



Basics of the x86-64 ISA with extensions

SIMD instructions are distinguished by:

VEX/EVEX prefix:	**`v`**
Operation:	**`mul, add, fmadd, mov`**
Modifier:	nontemporal (**`nt`**), unaligned (**`u`**), aligned (**`a`**), high (**`h`**), low (**`l`**)
Width:	scalar (**`s`**), packed (**`p`**)
Data type:	single (**`s`**), double (**`d`**)

… and many more

Examples:

`vmulpd`	→	Multiply Packed Double-Precision Floating-Point Values
`vfmadd213ps`	→	Fused Multiply-Add of Packed Single-Precision Floating-Point Values
`addsd`	→	Add Scalar Double-Precision Floating-Point Values
`vmovntdq`	→	Store Packed Integers Using Non-Temporal Hint

Case Study: Sum reduction (DP)

```
double sum = 0.0;

for (int i=0; i<size; i++){
    sum += data[i];
}
```

To get object code use **`objdump -d`** on object file or executable or compile with **`-S`**

Assembly code w/ **`-O1`** (AT&T syntax, Intel compiler):

```
.label:
    addsd   0(%rdi, %rax, 8),%xmm0
    inc     %rax
    cmp     %rsi, %rax
    jl      .label
```

Intel syntax:

```
addsd   xmm0, [rdi + rax * 8]
```

Sum reduction (DP) – AVX512

Assembly code w/ **`-O3 -xCORE-AVX512 -qopt-zmm-usage=high`** :

```
..B3.28:
        vaddpd    (%r13,%rcx,8), %zmm5, %zmm5
        vaddpd    64(%r13,%rcx,8), %zmm4, %zmm4
        vaddpd    128(%r13,%rcx,8), %zmm3, %zmm3
        vaddpd    192(%r13,%rcx,8), %zmm2, %zmm2
        addq      $32, %rcx
        cmpq      %rbx, %rcx
        jb        ..B3.28
..B3.29:
        vaddpd    %zmm4, %zmm5, %zmm4
        vaddpd    %zmm2, %zmm3, %zmm2
        vaddpd    %zmm2, %zmm4, %zmm5
# [... SNIP ...]
..B3.34:
        vshuff32x4 $238, %zmm5, %zmm5, %zmm2
        vaddpd    %zmm5, %zmm2, %zmm3
        vpermpd   $78, %zmm3, %zmm4
        vaddpd    %zmm4, %zmm3, %zmm5
        vpermpd   $177, %zmm5, %zmm6
        vaddpd    %zmm6, %zmm5, %zmm7
        vaddsd    %xmm1, %xmm7, %xmm1
```

Remainder omitted

Bulk loop code
(8x4-way unrolled)

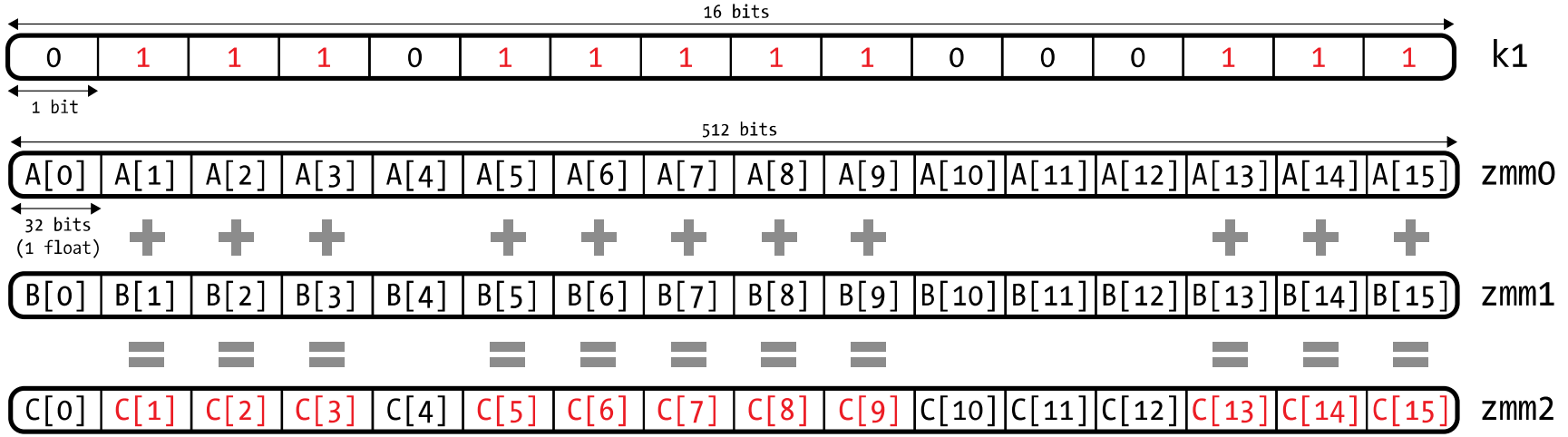
Sum up 32
partial sums into
xmm1

Example for masked execution

Masking is very helpful in cases such as, e.g., remainder loop handling or conditionals

Available on x86 starting with AVX-512

Example: **`vaddps %zmm0, %zmm1, %zmm2{%k1}`**



SIMD with masking – sum reduction with condition

```
double sum = 0.0;

for (int i=0; i<size; i++){
    if (data[i] > 0.0)
      sum += data[i];
}
```

Bulk loop code (8x4-way unrolled)

```
..B1.38:
vmovups   (%r15,%rcx,8), %zmm6
vmovups   64(%r15,%rcx,8), %zmm7
vmovups   128(%r15,%rcx,8), %zmm8
vmovups   192(%r15,%rcx,8), %zmm9
vcmppd    $14, %zmm10, %zmm6, %k1
vcmppd    $14, %zmm10, %zmm7, %k2
vcmppd    $14, %zmm10, %zmm8, %k3
vcmppd    $14, %zmm10, %zmm9, %k4
vaddpd    %zmm6, %zmm5, %zmm5{%k1}
vaddpd    %zmm7, %zmm4, %zmm4{%k2}
vaddpd    %zmm8, %zmm3, %zmm3{%k3}
vaddpd    %zmm9, %zmm2, %zmm2{%k4}
addq      $32, %rcx
cmpq      %r14, %rcx
jb        ..B1.38
```

SIMD mask generation

masked SIMD ADDs (accumulates)

NHR FAU

Friedrich-Alexander-Universität
Erlangen-Nürnberg

STREAM Triad

A pen & paper in-core analysis

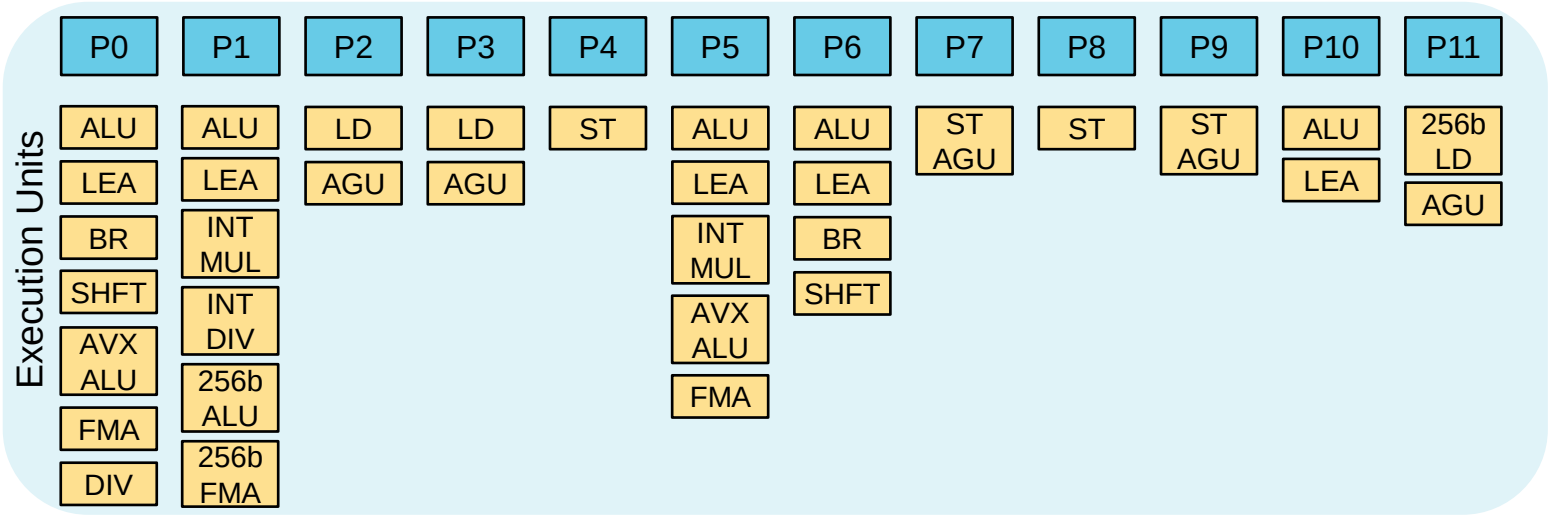


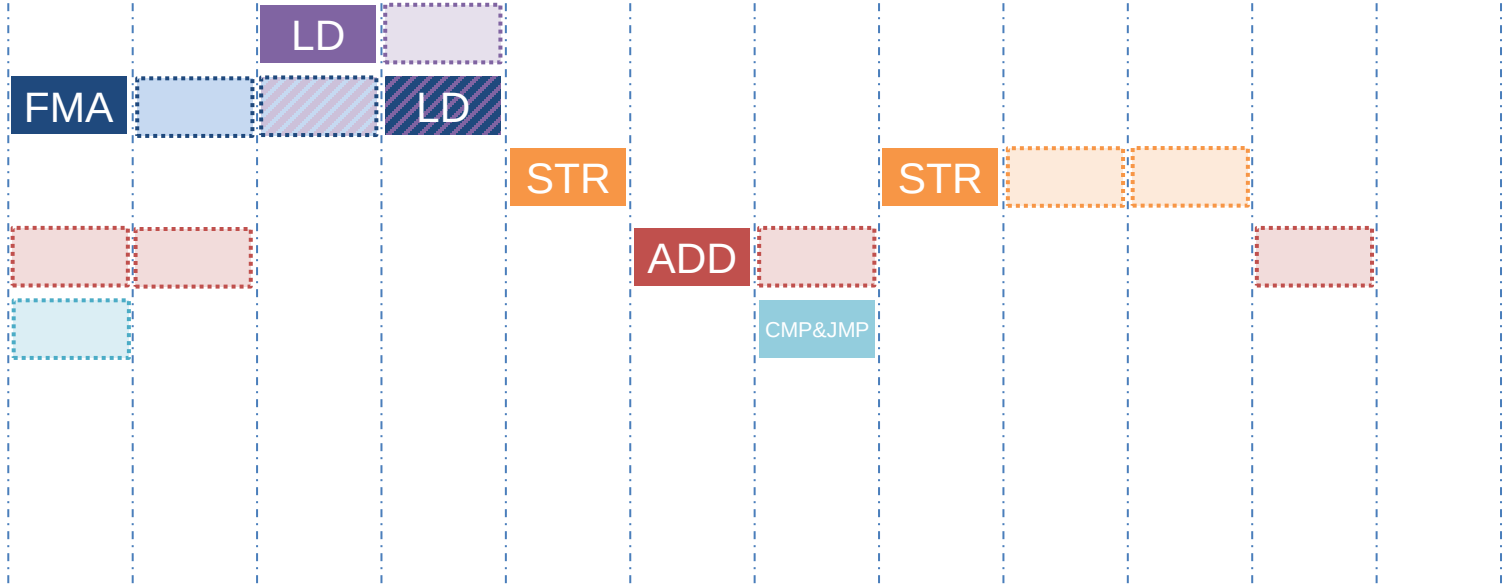
STREAM TRIAD on Intel Sapphire Rapids

STREAM TRIAD

```
a[i] = b[i] + s * c[i]
```

```
..B2.42:
  vmovups      (%r14,%rdx,8), %zmm1
  vfmadd213pd  (%r15,%rdx,8), %zmm2, %zmm1
  vmovupd      %zmm1, (%r12,%rdx,8)
  addq         $8, %rdx
  cmpq         %rsi, %rdx
  jb           ..B2.42
```



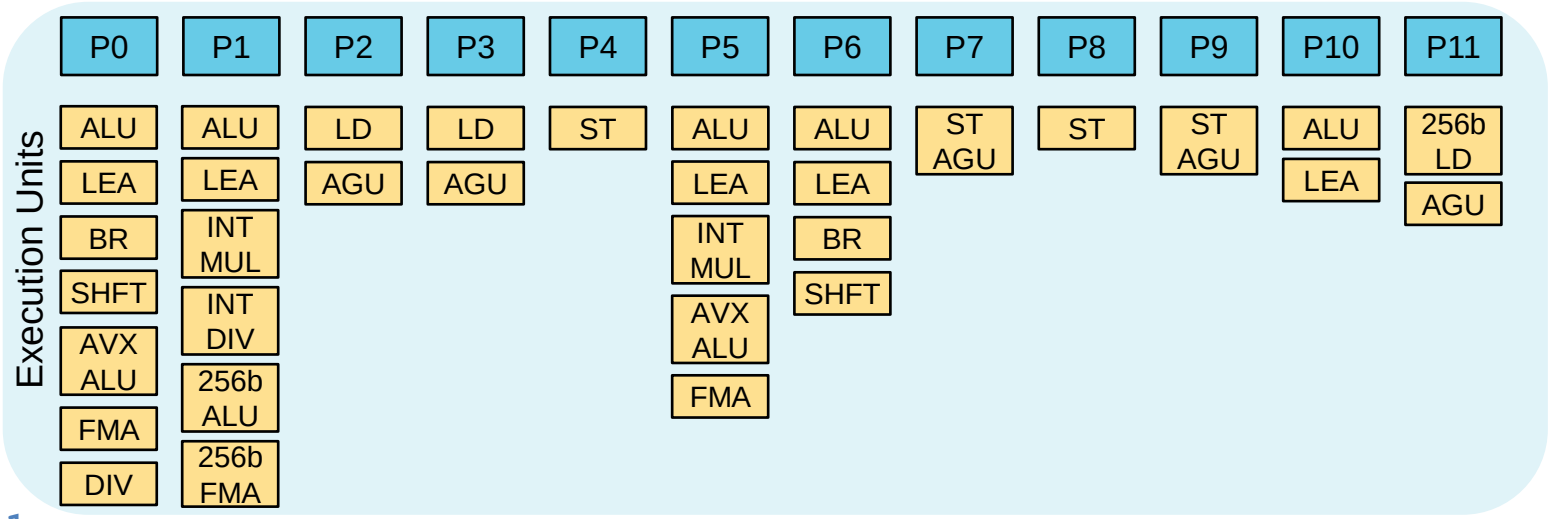


STREAM TRIAD on Intel Sapphire Rapids

STREAM TRIAD

```
a[i] = b[i] + s * c[i]
```

```
..B2.42:
  vmovups       (%r14,%rdx,8), %zmm1
  vfmadd213pd   (%r15,%rdx,8), %zmm2, %zmm1
  vmovupd       %zmm1, (%r12,%rdx,8)
  addq          $8, %rdx
  cmpq          %rsi, %rdx
  jb            ..B2.42
```





1 cy / 8 it

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Hands-On #1: Dot product

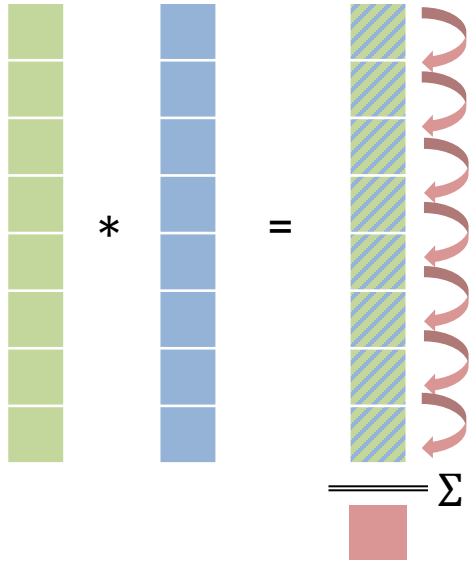
➔ https://go-nhr.de/CLPE-ex1



Hands-On: Benchmarking Dot Product

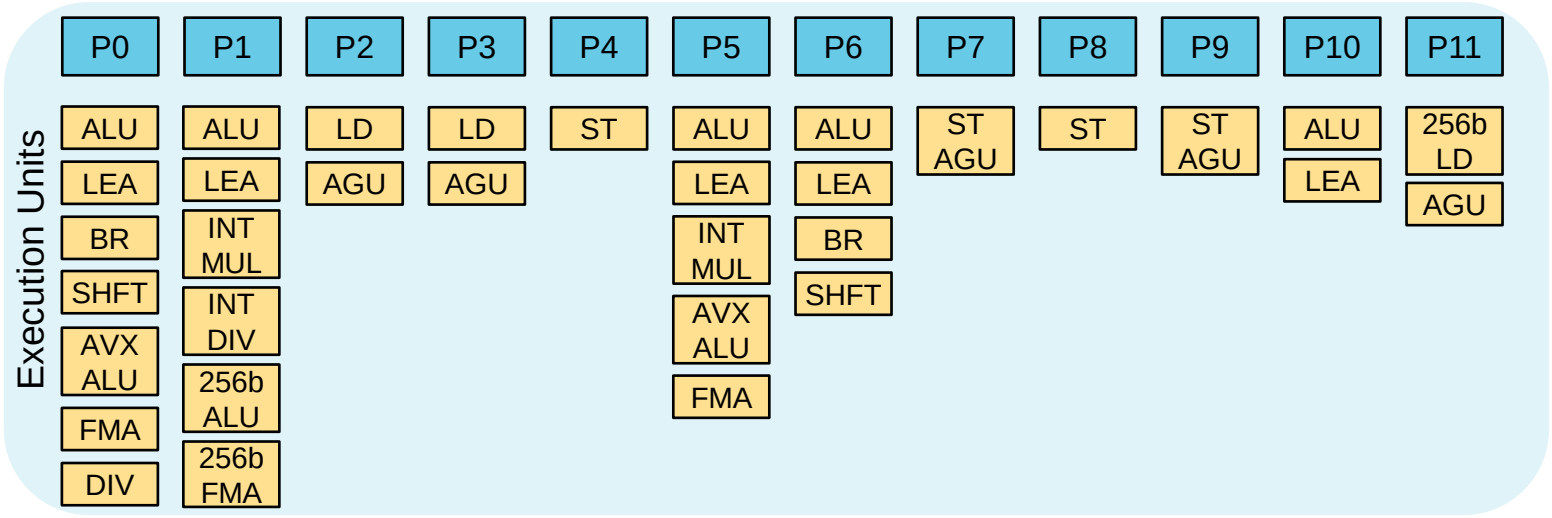
Dot product

```
s = s + a[i] * b[i]
```



→ **Moodle, hands-on #1**

Dot Product on SPR





NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Hands-On #2:
Dot product (with Compiler Explorer)

➔ https://go-nhr.de/CLPE-ex2



Dot Product on SPR – CE view

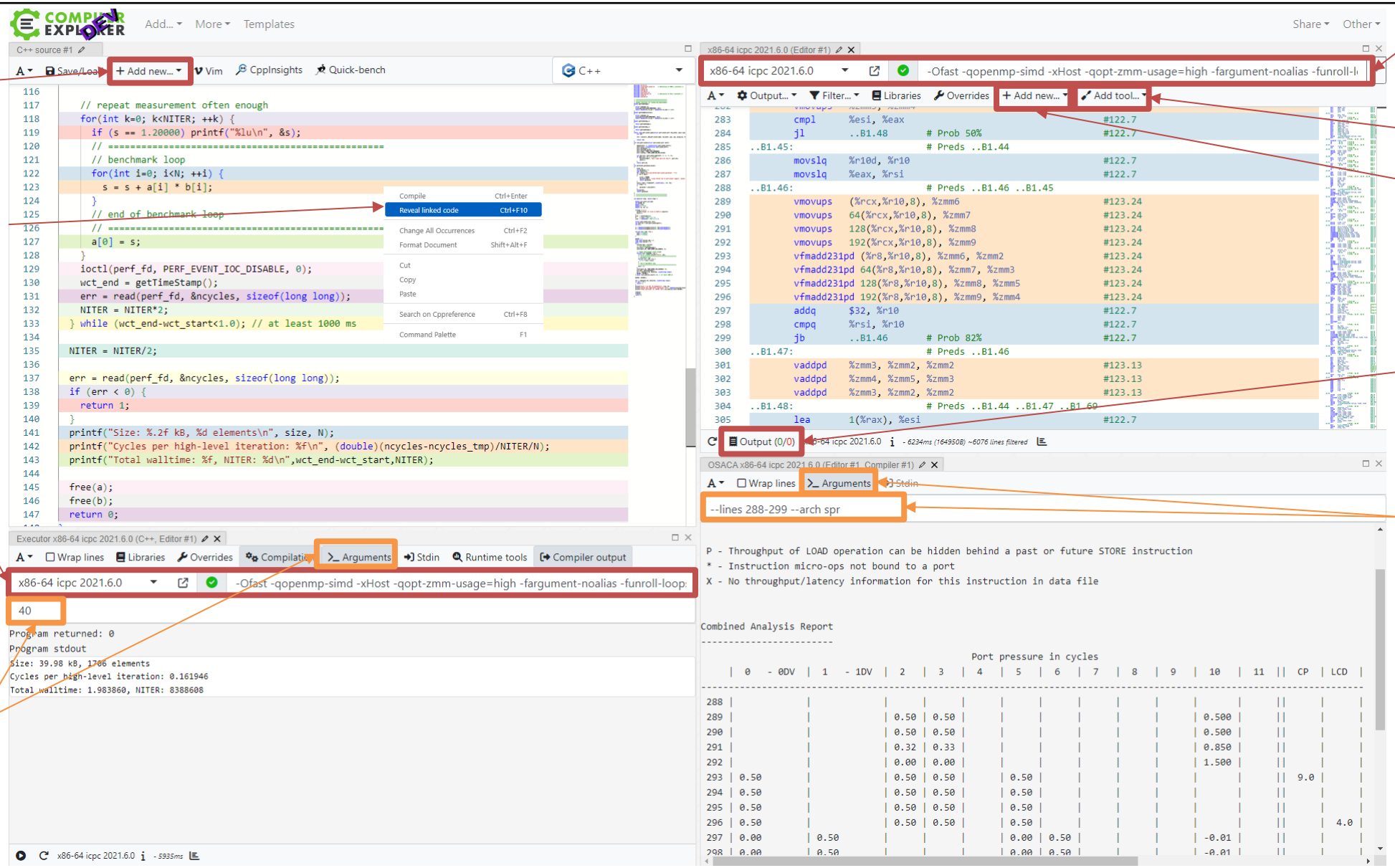
→ **Moodle, hands-on #2**

Add new compiler

Right click and “Reveal linked code” can help you find your region of interest

Set **executer** compiler and flags (separately from ASM compiler)

Set **runtime parameters**



Set **ASM compiler** compiler and flags

Add new **analysis tool**

Add new **executor**

Click to see your **compiler log** (warnings and errors)

Set **OSACA runtime parameters**

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

The Open-Source Architecture Code Analyzer (OSACA)

An introduction



OSACA

- Open Source Architecture Code Analyzer
- Static in-core code analysis

  Assumptions

  - Steady-state execution (no warm-up/cool-down)
  - All data in L1
  - Perfect out-of-order scheduling
  - (currently) no front-end, i.e., no limit in instruction fetching, decoding, etc…
- Architecture specific model for each µArch

- Python module

```
$ pip install osaca
```

OSACA – Usage

```
osaca [-h] [-V] [--arch ARCH] [--fixed] [--lines LINES]
      [--ignore-unknown] [--lcd-timeout SECONDS]
      [--db-check] [--import MICROBENCH] [--insert-marker]
      [--export-graph GRAPHNAME] [--consider-flag-deps]
      [--out OUT] [--verbose]
      FILEPATH
```

Important flags:

`--arch ARCH` Currently supported: Intel **SNB** – **GNR**, AMD **ZEN1**, **ZEN2**, **ZEN3**, **ZEN4**, **ZEN5**, Arm **TX2**, **A72**, **N1**, **A64FX**, **TSV110**, **M1**, **V2 (Grace)**

`--lines L1,L2,L3-L4,L5:L6` Specify lines to analyze (if no markers are used)

`--ignore-unknown` Assume **0cy** TP/LAT for unknown instructions

Marking the region of interest

Comment marker

x86

```
# OSACA-BEGIN
.L22:
  vmovapd 0(%r13,%rax),%ymm0
  vfmadd213pd (%r14,%rax),%ymm1,%ymm0
  vmovapd %ymm0,(%r12,%rax)
  addq $32,%rax
  cmpq %rax,%r15
  jne .L22
# OSACA-END
```

arm

```
// OSACA-BEGIN
.L18:
  ldr q2, [x20, x0]
  ldr q1, [x21, x0]
  fmla v1.2d, v2.2d, v0.2d
  str q1, [x19, x0]
  add x0, x0, #16
  cmp x22, x0
  bne .L18
// OSACA-END
```

Insertion tool

```
$ osaca --arch ARCH --insert-marker

Blocks found in assembly file:
.L_A
          ...
.L_B
          ...
----------------------------
Possible blocks to be marked:
.L_A
.L_B
Choose block to be marked [.L_B]: _
```

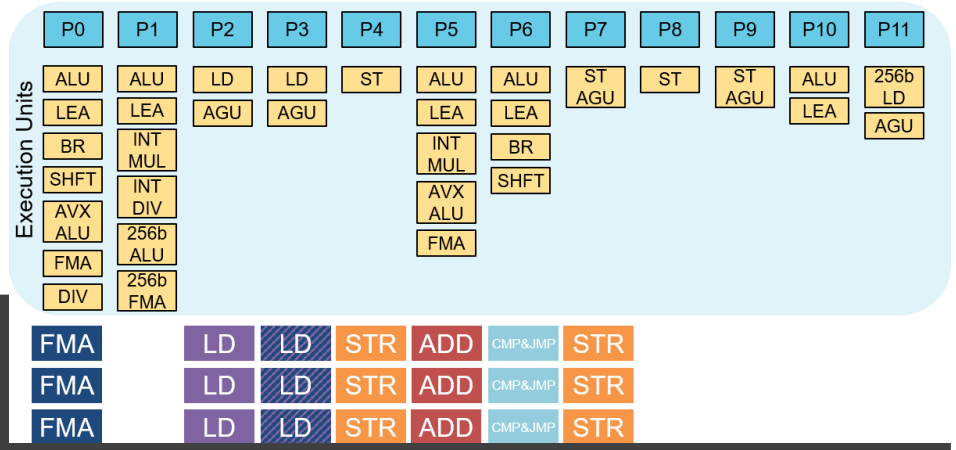
will be marked with **byte** markers, i.e.:

```
movl $111,%ebx; .byte 100,103,144;   (x86)
...
movl $222,%ebx; .byte 100,103,144;
```

```
mov x1,#111; .byte 213,3,32,31   (aarch64)
...
mov x1,#222; .byte 213,3,32,31
```

Triad on SPR with OSACA

- Recap: Manual analysis resulted in 1 cy/8 it




```
$ osaca --arch SPR triad.s
Open Source Architecture Code Analyzer (OSACA) - 0.6.0
Architecture:       SPR

 * - Instruction micro-ops not bound to a port
 X - No throughput/latency information for this instruction in data file

                                    Port pressure in cycles
  | 0   - 0DV | 1   - 1DV|  2   |  3   |  4   |  5   |  6   |  7   |  8   |  9   |  10  |  11  ||CP |LCD|
--------------------------------------------------------------------------------------------------------
2 |           |          |      |      |      |      |      |      |      |      |      |      ||   |   |  ..B2.42:
3 |           |          | 0.50 | 0.50 |      |      |      |      |      |      | 0.50 || 5 |   |  vmovups (%r14,%rdx,8), %zmm1
4 | 0.50      |          | 0.50 | 0.50 |      | 0.50 |      |      |      |      | 0.50 || 4 |   |  vfmadd213pd (%r15,%rdx,8), %zmm2, %zmm1
5 |           |          |      |      | 1.00 |      |      | 1.00 | 1.00 | 1.00 |      |      || 0 |   |  vmovupd     %zmm1, (%r12,%rdx,8)
6 | 0.10      | 0.26     |      |      |      | 0.10 | 0.27 |      |      |      | 0.27 |      ||   | 1 |  addq        $8, %rdx
7 | 0.00      | 0.34     |      |      |      | 0.00 | 0.33 |      |      |      | 0.33 |      ||   |   |  cmpq        %rsi, %rdx
8 |           |          |      |      |      |      |      |      |      |      |      |      ||   |   |* jb          ..B2.42
--------------------------------------------------------------------------------------------------------
    0.60        0.60       1.00   1.00   1.00   0.60   0.60   1.00   1.00   1.00   0.60   1.00    9   1


Loop-Carried Dependencies Analysis Report
-----------------------------------------
   6 |  1.0 | addq        $8, %rdx                | [6]
```

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Hands-On #3: Dot Product with OSACA

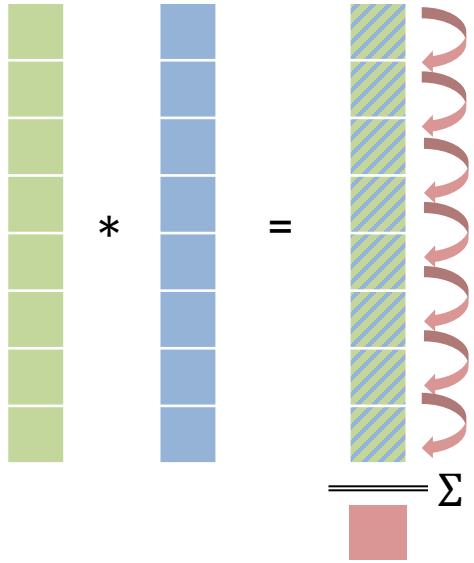
➔ https://go-nhr.de/CLPE-ex3



Hands-On: Benchmarking Dot Product (DP)

Dot Product

```
s = s + a[i] * b[i]
```



→ **Moodle, hands-on #3**

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Hands-On #4: PI by integration

➔ https://go-nhr.de/CLPE-ex4



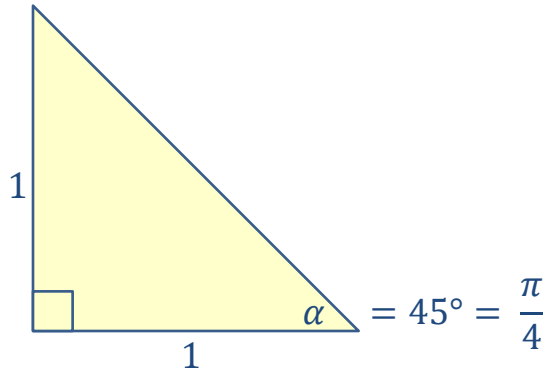
Hands-On: PI by integration

PI

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

```
double delta_x = 1./n;
double sum     = 0.0;

for (int i=0; i<n; i++)
{
  x    = (i + 0.5) * delta_x;
  sum += (4.0 / (1.0 + x * x));
}
```



$= 45° = \frac{\pi}{4}$

$$\tan(\alpha) = \frac{1}{1} = 1 \;\Rightarrow\; \arctan(1) = \frac{\pi}{4}$$

$$\Rightarrow \pi = 4 \cdot \arctan(1)$$

$$\frac{d}{dx}\arctan(x) = \frac{1}{1+x^2} \Rightarrow \pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

→ **Moodle, hands-on #4**

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

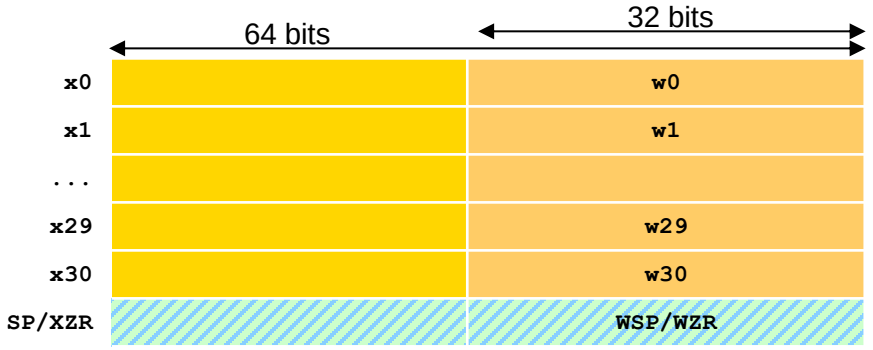
Aarch64 ISA



AArch64 ISA – differences to x86

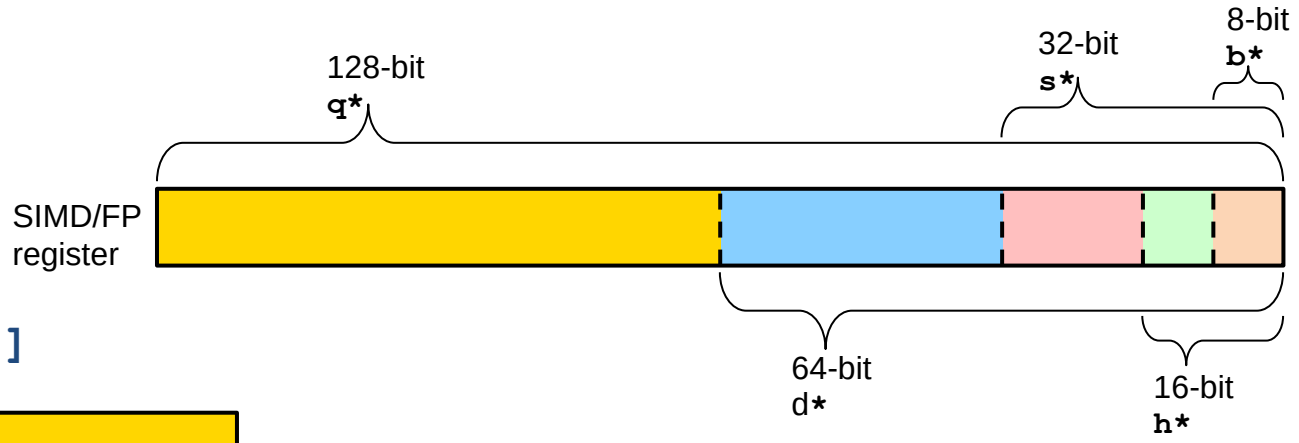
- Opcodes are always 32 bits
- Similar to Intel (left) syntax with STORE (**STR/STP**) as exception
  - `add  x1, x1, 8     # x1 ← x1 + 8`
  - `ldr  x0, [x1]      # x0 ← mem at x1`
  - `ldp  x0, x1, [x2]  # x0, x1 ← mem at x2`
  - `str  x0, [x1]      # mem at x1 ← x0`
  - `stp  x0, x1, [x2]  # mem at x2 ← x0, x1`

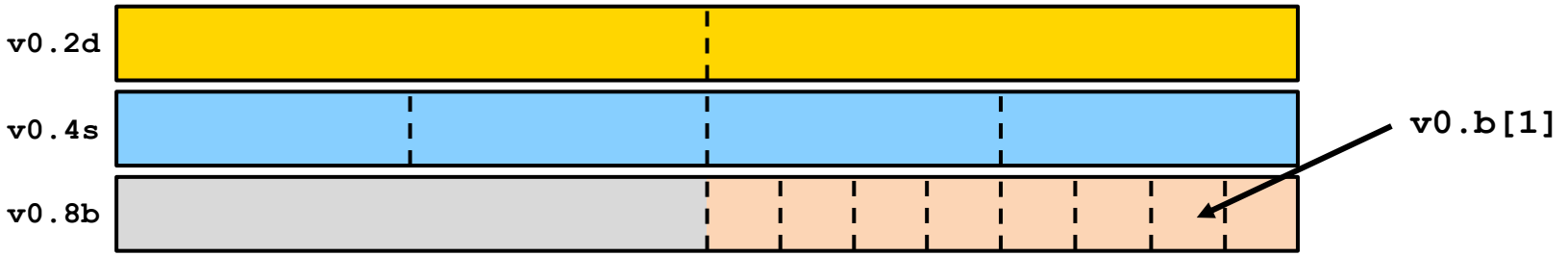
- 31 general purpose registers (64 bits):
  - **x0–x30** (aliases with 32-bit GPRs **w0–w30**)
  - 32nd register is stack pointer and zero register



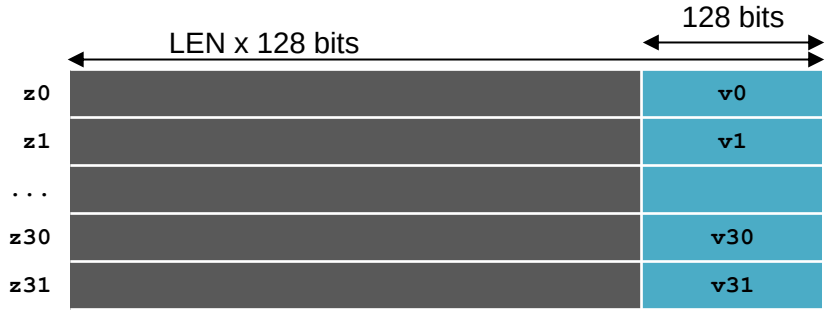
AArch64 ISA – differences to x86

- 32 SIMD and FP registers (NEON, 128 bits)
  - **`v0–v31`**
  - can be optionally specified with shapes and lanes **`vn.<LANES><SHAPE>`**
  - a single element can be indexed via brackets **`[i]`**





- 32 scalable vector registers (128–2048 bits):
  - **`z0–z31`**, extending **`v`** registers, multiples of 128 bits
  - size defined in OS
- 16 predicate registers (16–256 bits)
  - **`p0–p15`**, multiples of 16 bits
  - optional with predication operation **`/z`**, **`/m`**, **`/x`** (zeroing, merging, don't care)

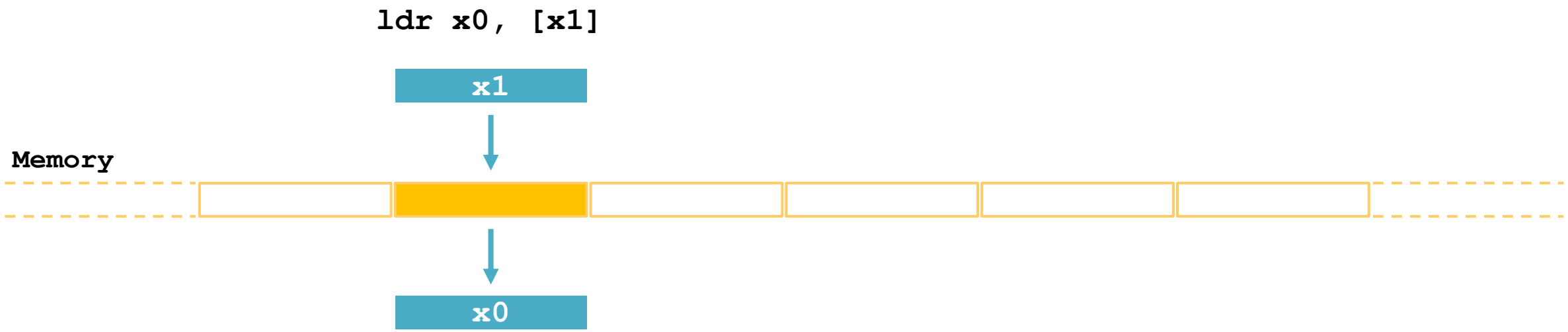


AArch64 ISA – differences to x86

- Addressing Modes:
  - Simple (`[BASE]`) — `ldr x0, [x1]`
  - Offset (`[BASE, OFFSET]`) — `ldr x0, [x1, #64]`
  - Modified Offset — `ldr x0, [x1, x2, lsl 3]`
  - Pre-indexed (`[BASE, OFFSET]!`) — `ldr x0, [x1, #64]!`
  - Post-indexed (`[BASE], OFFSET`) — `ldr x0, [x1], #64`

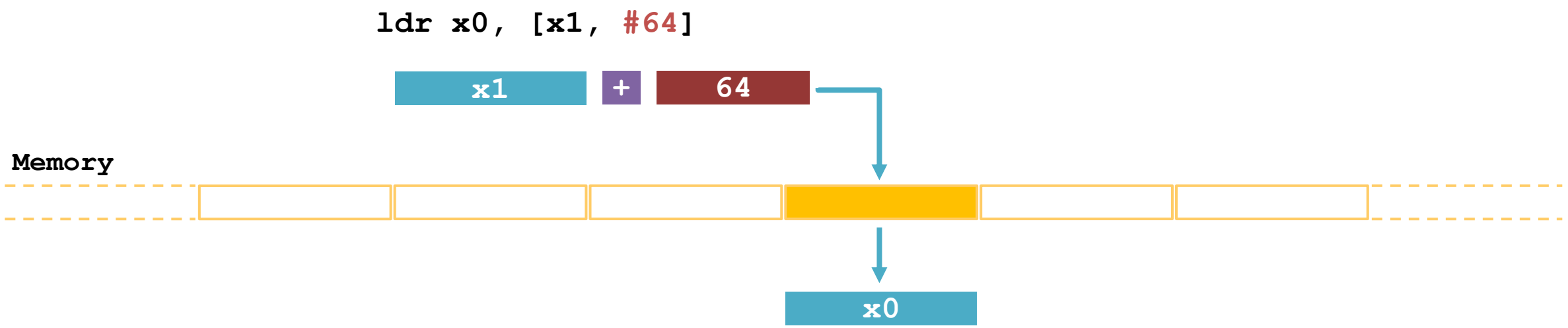
AArch64 ISA – differences to x86

- Addressing Modes:
  - Simple (`[BASE]`) — `ldr x0, [x1]`
  - Offset (`[BASE, OFFSET]`) — `ldr x0, [x1, #64]`
  - Modified Offset — `ldr x0, [x1, x2, lsl 3]`
  - Pre-indexed (`[BASE, OFFSET]!`) — `ldr x0, [x1, #64]!`
  - Post-indexed (`[BASE], OFFSET`) — `ldr x0, [x1], #64`



AArch64 ISA – differences to x86

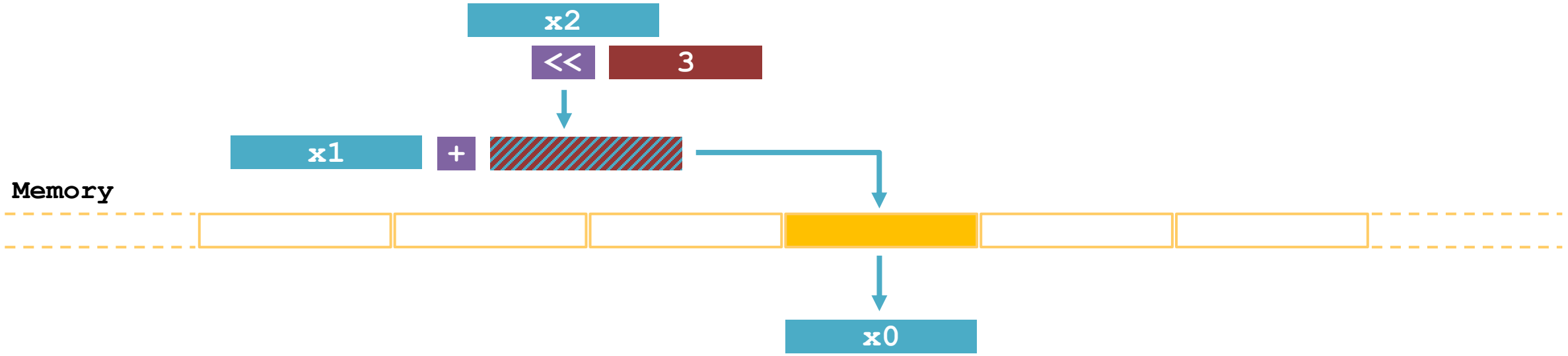
- Addressing Modes:
  - Simple (`[BASE]`) — `ldr x0, [x1]`
  - Offset (`[BASE, OFFSET]`) — `ldr x0, [x1, #64]`
  - Modified Offset — `ldr x0, [x1, x2, lsl 3]`
  - Pre-indexed (`[BASE, OFFSET]!`) — `ldr x0, [x1, #64]!`
  - Post-indexed (`[BASE], OFFSET`) — `ldr x0, [x1], #64`



AArch64 ISA – differences to x86

- Addressing Modes:
  - Simple (`[BASE]`) — `ldr x0, [x1]`
  - Offset (`[BASE, OFFSET]`) — `ldr x0, [x1, #64]`
  - Modified Offset — `ldr x0, [x1, x2, lsl 3]`
  - Pre-indexed (`[BASE, OFFSET]!`) — `ldr x0, [x1, #64]!`
  - Post-indexed (`[BASE], OFFSET`) — `ldr x0, [x1], #64`

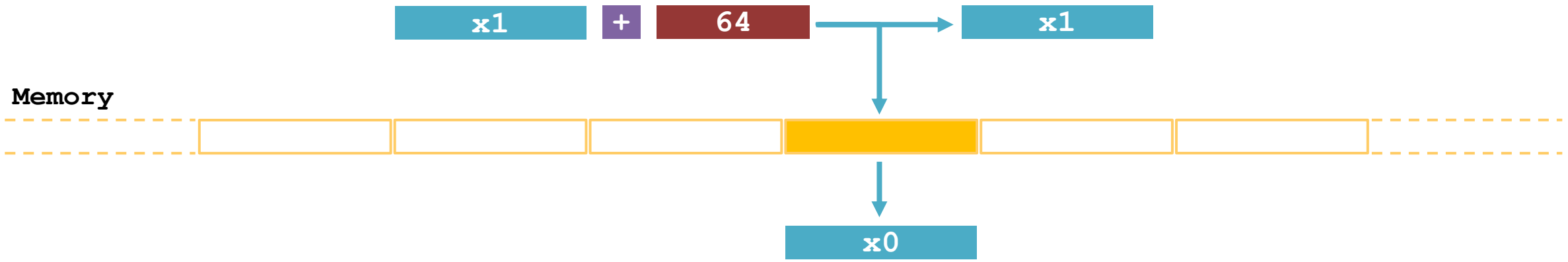
`ldr x0, [x1, x2, lsl 3]`



AArch64 ISA – differences to x86

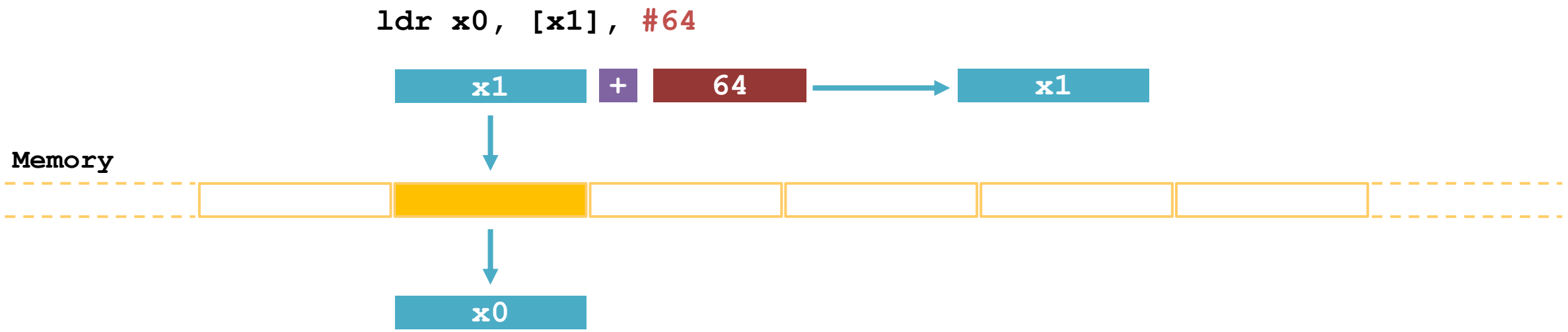
- Addressing Modes:
  - Simple (`[BASE]`) — `ldr x0, [x1]`
  - Offset (`[BASE, OFFSET]`) — `ldr x0, [x1, #64]`
  - Modified Offset — `ldr x0, [x1, x2, lsl 3]`
  - Pre-indexed (`[BASE, OFFSET]!`) — `ldr x0, [x1, #64]!`
  - Post-indexed (`[BASE], OFFSET`) — `ldr x0, [x1], #64`

`ldr x0, [x1, #64]!`



AArch64 ISA – differences to x86

- Addressing Modes:
  - Simple (`[BASE]`) — `ldr x0, [x1]`
  - Offset (`[BASE, OFFSET]`) — `ldr x0, [x1, #64]`
  - Modified Offset — `ldr x0, [x1, x2, lsl 3]`
  - Pre-indexed (`[BASE, OFFSET]!`) — `ldr x0, [x1, #64]!`
  - Post-indexed (`[BASE], OFFSET`) — `ldr x0, [x1], #64`



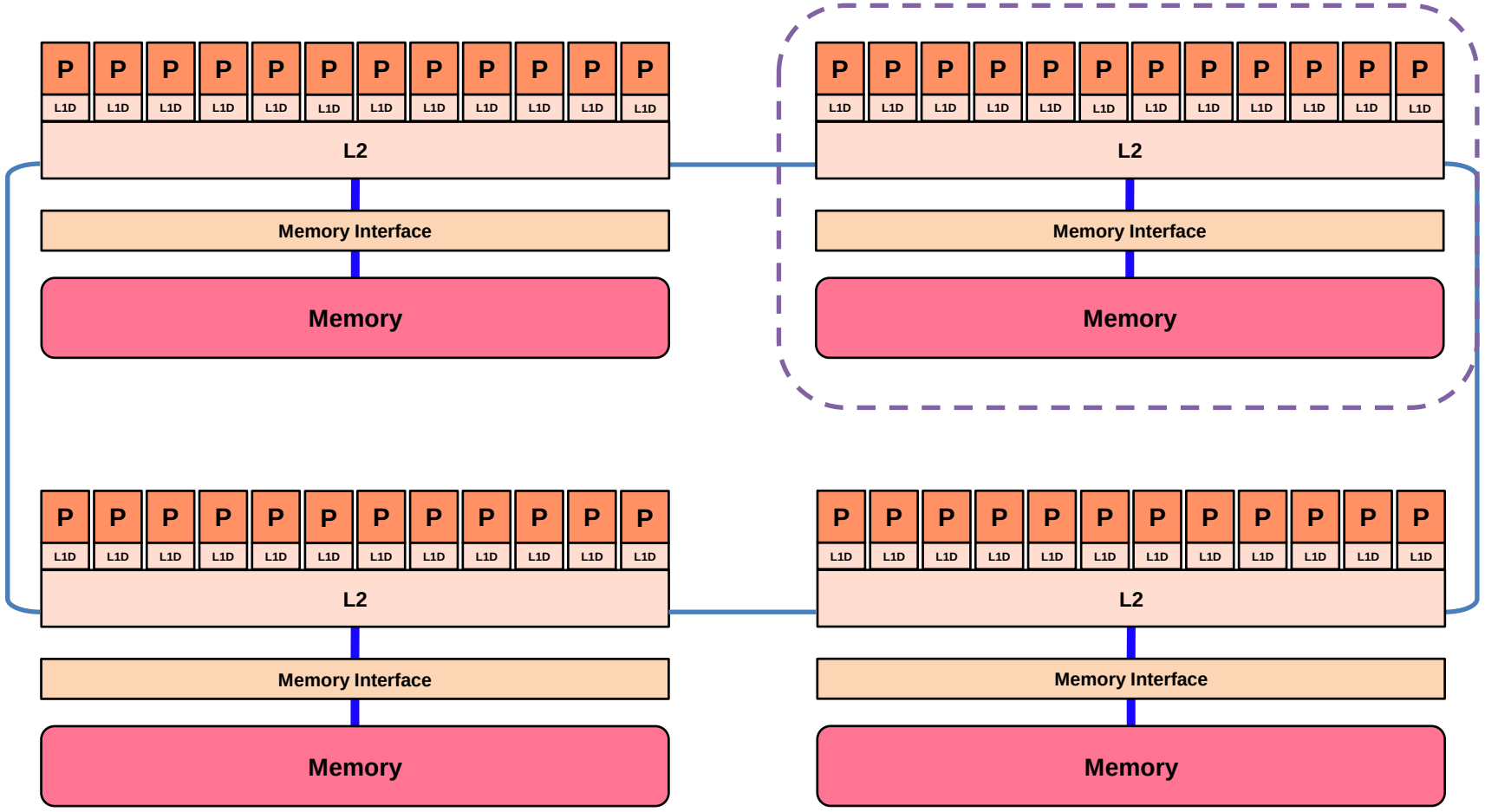
NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

A64FX core architecture and AArch64 Arm ISA



Node architecture of A64FX – FX700



1 Core Memory Group (CMG)

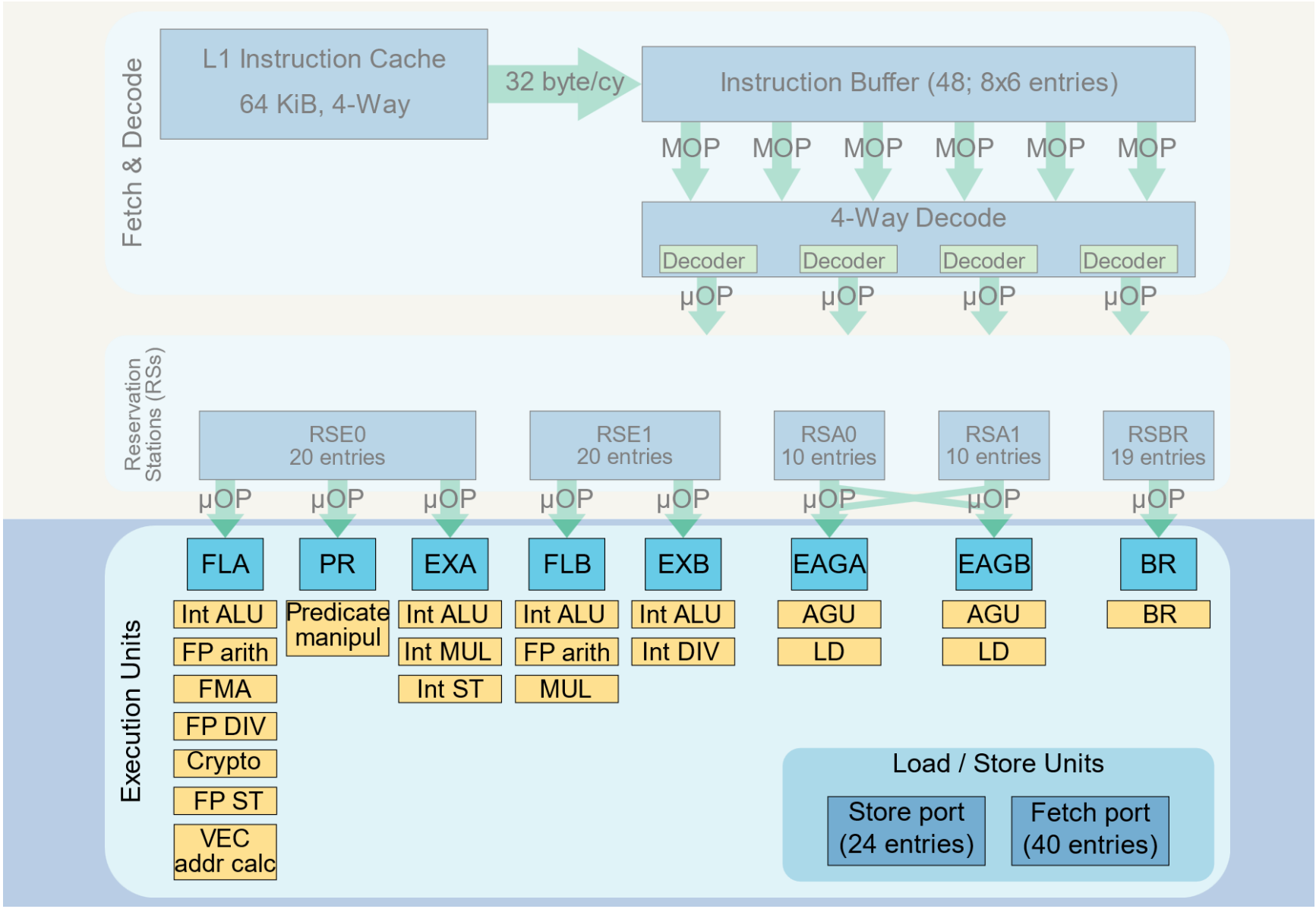
4 CMGs per node

48 cores per node

Port model for the A64FX

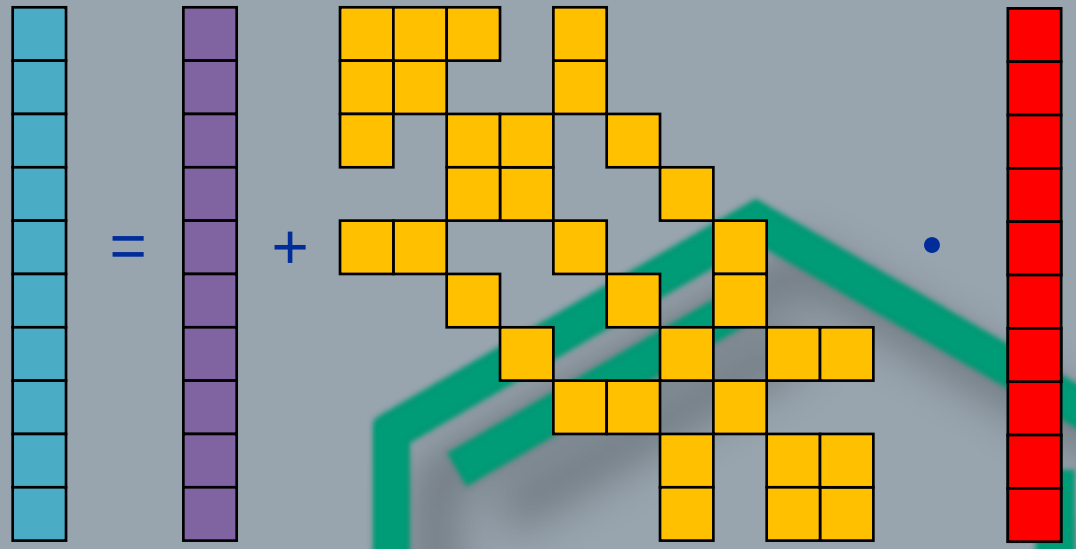
Frontend

Backend



Case Study: SpMV on A64FX

Sparse Matrix-Vector Multiplication

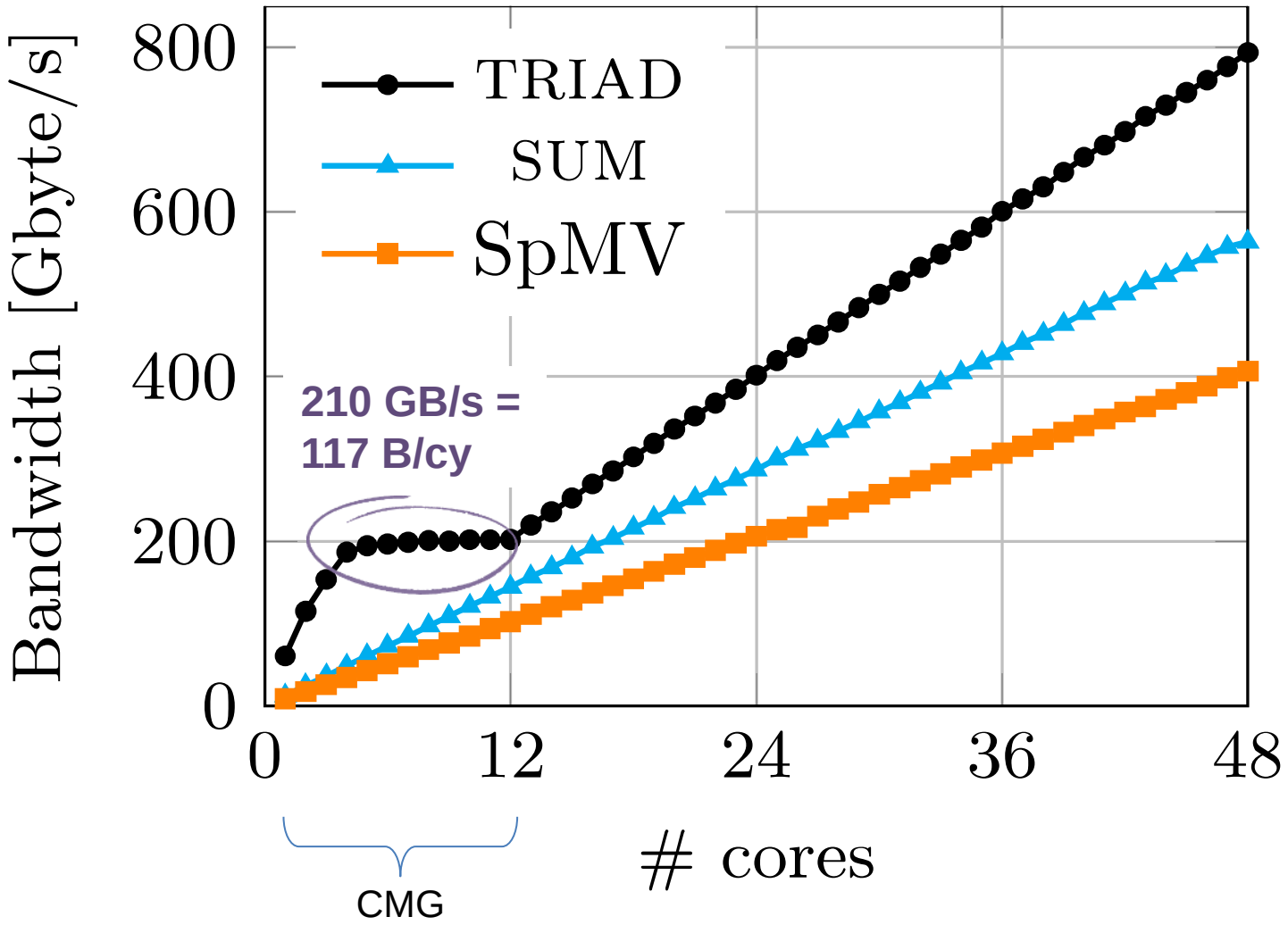


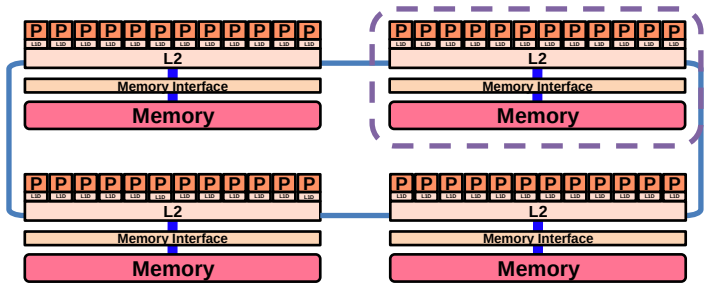
Based on:

C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig:
ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.
Concurrency and Computation: Practice and Experience, e6512 (2021).

DOI: 10.1002/cpe.6512

Motivation



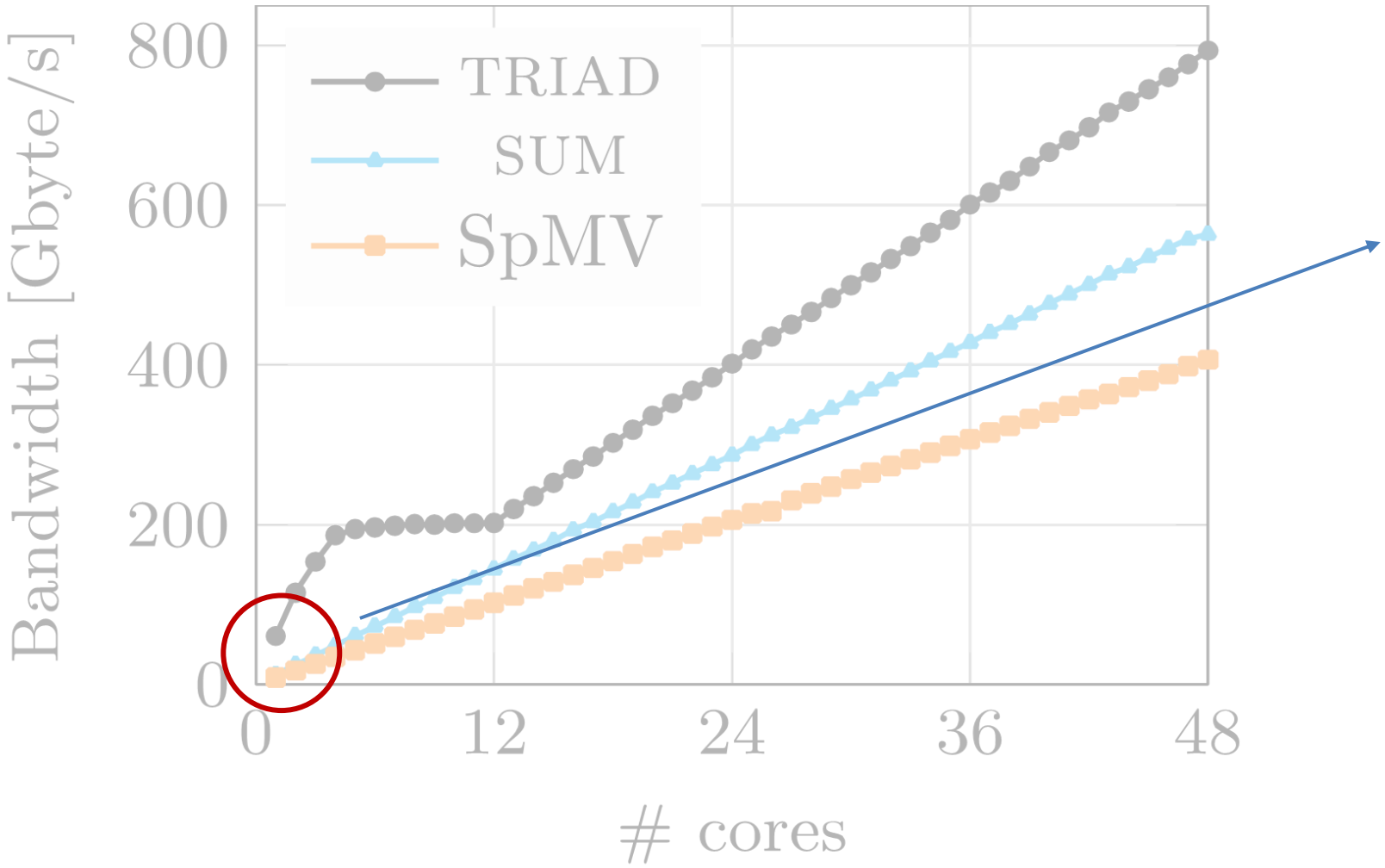


Clear memory bandwidth saturation for STREAM TRIAD (`a[i] = b[i] + s*c[i]`)

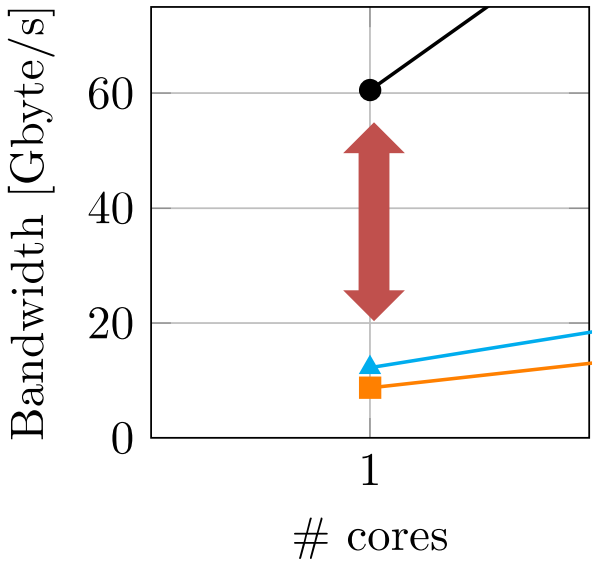
But why not for SUM (`s += a[i]`) and SpMV (`b[:] = A[:,:]*x[i[:]]`)?

Thread pinning : Compact

Motivation



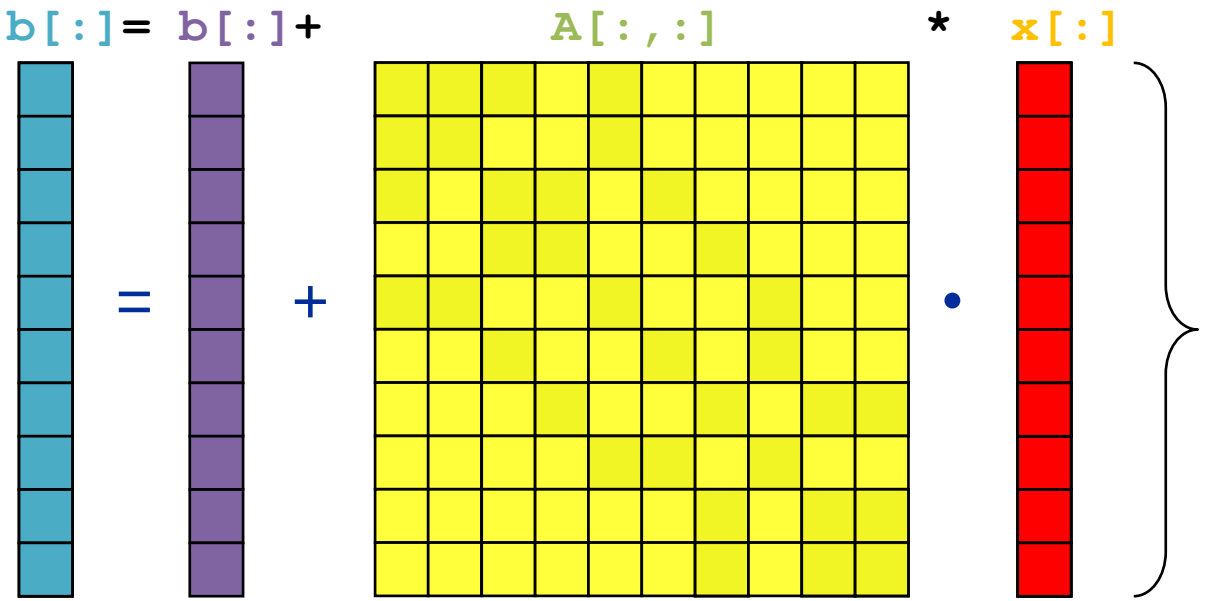
Thread pinning : Compact



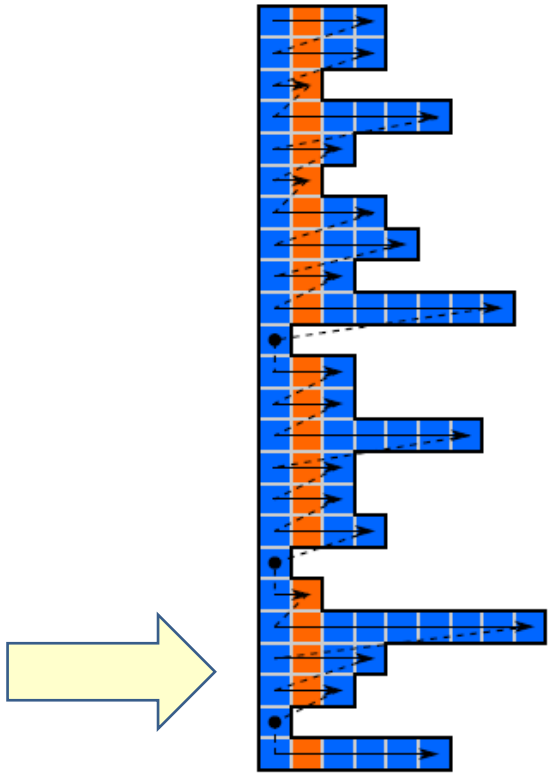
Understanding single-core performance is the key!

SpMV

Sparse Matrix-Vector Multiplication (SpMV) : `b=Ax`



General case: some indirect addressing required!



In Compressed Row Storage (CRS) format

```
for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
        b[i] = b[i] + A[j] * x[col_idx[j]]
```

SpMV

Assembly of the short inner-loop

```
.L6:
        ld1sw   z0.d, p0/z, [x17, x20, lsl 2]
        ld1d    z2.d, p0/z, [x18, x20, lsl 3]
        ld1d    z3.d, p0/z, [x30, z0.d, lsl 3]
        add     x20, x20, 8
        fmla    z1.d, p0/m, z3.d, z2.d
        whilelo p0.d, x20, x14
        b.any   .L6

        faddv   d4, p1, z1.d
```

load **col_idx[j]**

load **A[j]**

gather from **x[ ]**

increase loop counter **j**

"the work"

loop mechanics

reduction across SIMD register

In Compressed Row Storage (CRS) format

```
for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
        b[i] = b[i] + A[j] * x[col_idx[j]]
```

SpMV

Assembly of the short inner-loop

```
.L6:
        ld1sw   z0.d, p0/z, [x17, x20, lsl 2]
        ld1d    z2.d, p0/z, [x18, x20, lsl 3]
        ld1d    z3.d, p0/z, [x30, z0.d, lsl 3]
        add     x20, x20, 8
        fmla    z1.d, p0/m, z3.d, z2.d
        whilelo p0.d, x20, x14
        b.any   .L6

        faddv   d4, p1, z1.d
```

FMA: Update **z1.d**

Latency: 9 cycles

Loop length : 27
HPCG matrix

Horizontal add of 512-bit register

latency = 49 cycles

85 cy per inner loop traversal
→ 100 GB/s per CMG

→ No saturation



In Compressed Row Storage (CRS) format

```
for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
        b[i] = b[i] + A[j] * x[col_idx[j]]
```

SpMV

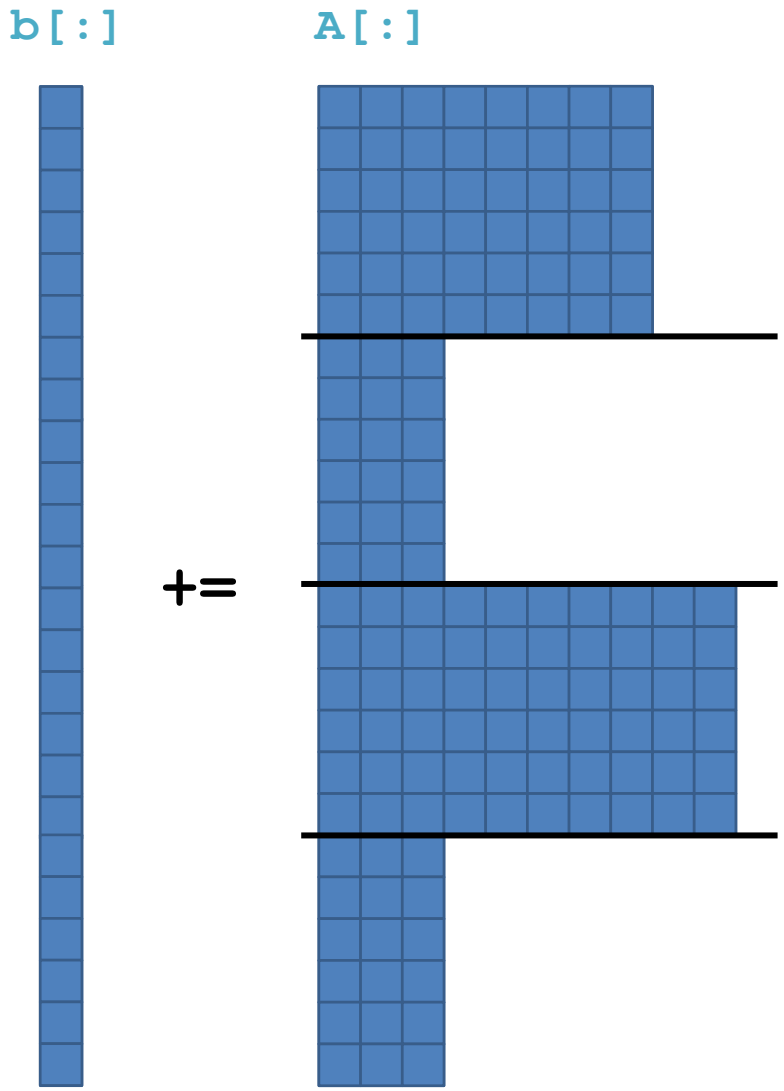
```
$ osaca --arch a64fx spmv-inner-loop.s
[...]


Combined Analysis Report
------------------------
                                Port pressure in cycles
 | 0   - 0DV |  1   |  2   |  3   |  4   |  5  -  5D  |  6   -  6D  |  7   ||  CP  | LCD  |
-------------------------------------------------------------------------------------------
1|           |      |      |      |      |             |             |      ||      |      | .L6:
2|           |      |      |      |      | 0.50  0.50  | 0.50   0.50 |      ||  8.0 |      | ld1sw z0.d, p0/z, [x17,x20,lsl 2]
3|           |      |      |      |      | 0.50  0.50  | 0.50   0.50 |      ||      |      | ld1d z2.d, p0/z, [x18,x20,lsl 3]
4| 1.00      |      |      | 1.00 |      | 2.00  2.00  | 2.00   2.00 |      || 11.0 |      | ld1d z3.d, p0/z, [x30,z0.d,lsl 3]
5| 0.00      |      | 0.00 | 0.00 | 1.00 |             |             |      ||      |      | add  x20, x20, 8
6| 0.00      |      | 1.00 |      |      |             |             |      ||  9.0 |  9.0 | fmla z1.d, p0/m, z3.d, z2.d
7|           | 1.00 |      | 1.00 |      |             |             |      ||      |      | whilelo  p0.d, x20, x14
8|           |      |      |      |      |             |             | 1.00 ||      |      | b.any .L6

   1.00        1.00   1.00   2.00   1.00   3.00  3.00   3.00   3.00   1.00    28.0    9.0


Loop-Carried Dependencies Analysis Report
-----------------------------------------
   6 |  9.0 | fmla    z1.d, p0/m, z3.d, z2.d     | [6]
   5 |  1.0 | add     x20, x20, 8                | [5]
```

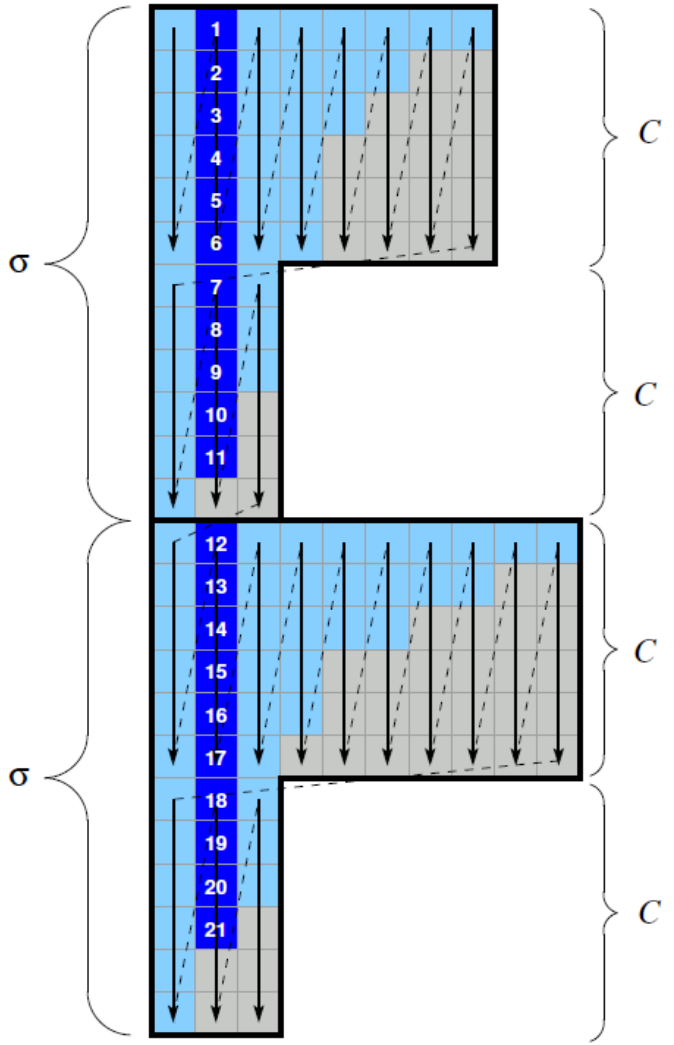
SIMD-friendly execution of SpMV with SELL-C-σ



- Inner loop goes down one block column
- Column-major storage within block
  → consecutive access of matrix
- Enables SIMD FMA instructions for column traversal in block and LHS update
- No reductions across SIMD register slots
- Longer inner loop (in assembly) than CRS
- RHS access still indirect (gather)

How to choose the parameters?

- C
  - $n \times$ SIMD width to allow good utilization of SIMD units
  - $n > 1$ useful for hiding ADD pipeline latency
- σ
  - As small as possible, as large as necessary
  - Large σ reduces zero padding
  - Sorting alters RHS access pattern



M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units*. SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). DOI: 10.1137/130930352,

SELL-32-σ kernel OSACA analysis for A64FX

	0 - 0DV	1	2	3	4	5 - 5D	6 - 6D	7	CP	LCD	
92											.L4:
93						0.50 0.50	0.50 0.50				ld1sw z16.d, p0/z, [x11]
94						0.50 0.50	0.50 0.50				ld1sw z17.d, p0/z, [x11, #1, mul vl]
95						0.50 0.50	0.50 0.50				ld1sw z20.d, p0/z, [x11, #2, mul vl]
96						0.50 0.50	0.50 0.50		8.0		ld1sw z21.d, p0/z, [x11, #3, mul vl]
97	0.00		0.00	0.00	1.00						add x10, x10, 32
98	0.00		0.00	0.00	1.00						add x11, x11, 128
99	0.00		0.00	0.00	1.00						add x12, x12, 256
100						0.50 0.50	0.50 0.50				ld1d z19.d, p0/z, [x12, #-4, mul vl]
101						0.50 0.50	0.50 0.50				ld1d z18.d, p0/z, [x12, #-3, mul vl]
102						0.50 0.50	0.50 0.50				ld1d z25.d, p0/z, [x12, #-2, mul vl]
103						0.50 0.50	0.50 0.50				ld1d z27.d, p0/z, [x12, #-1, mul vl]
104	1.00			1.00		2.00 2.00	2.00 2.00				ld1d z22.d, p0/z, [x3, z16.d, lsl 3]
105	1.00			1.00		2.00 2.00	2.00 2.00				ld1d z23.d, p0/z, [x3, z17.d, lsl 3]
106	1.00			1.00		2.00 2.00	2.00 2.00				ld1d z24.d, p0/z, [x3, z20.d, lsl 3]
107	1.00			1.00		2.00 2.00	2.00 2.00		11.0		ld1d z26.d, p0/z, [x3, z21.d, lsl 3]
108		1.00		1.00							whilelo p1.d, x10, x9
109	0.00		1.00								fmla z4.d, p0/m, z19.d, z22.d
110	0.00		1.00								fmla z5.d, p0/m, z18.d, z23.d
111	0.00		1.00								fmla z6.d, p0/m, z25.d, z24.d
112	0.00		1.00						.0	9.0	fmla z7.d, p0/m, z27.d, z26.d
113	0.00		0.00	0.00	1.00						mov p0.b, p1.b
114								1.00			b.any .L4
	4.00	1.00	4.00	5.00	4.00	12.0	12.0	1.00	28.0	9.0	

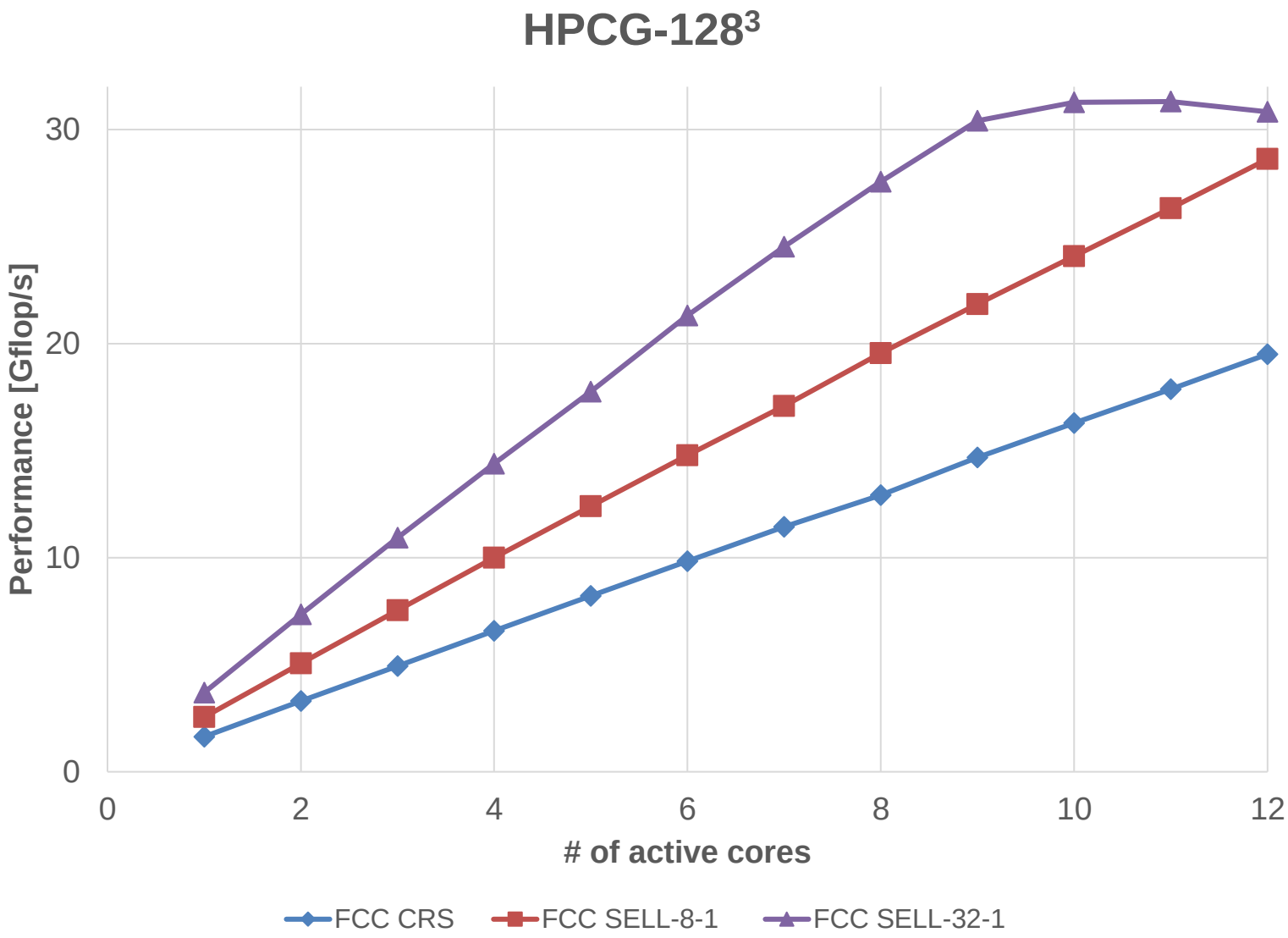
Shift of bottleneck

SELL-32-σ kernel OSACA analysis for A64FX

```
Loop-Carried Dependencies Analysis Report
-----------------------------------------
112 |   9.0 | fmla   z7.d, p0/m, z27.d, z26.d        | [112]
111 |   9.0 | fmla   z6.d, p0/m, z25.d, z24.d        | [111]
110 |   9.0 | fmla   z5.d, p0/m, z18.d, z23.d        | [110]
109 |   9.0 | fmla   z4.d, p0/m, z19.d, z22.d        | [109]
 99 |   1.0 | add    x12, x12, 256                    | [99]
 98 |   1.0 | add    x11, x11, 128                    | [98]
 97 |   1.0 | add    x10, x10, 32                     | [97]
```

SpMV performance with SELL-C-σ (1 CMG)

- SELL-C-σ separates SIMD from sum reduction
- C>8 allows for reduction of `fmla` latency impact



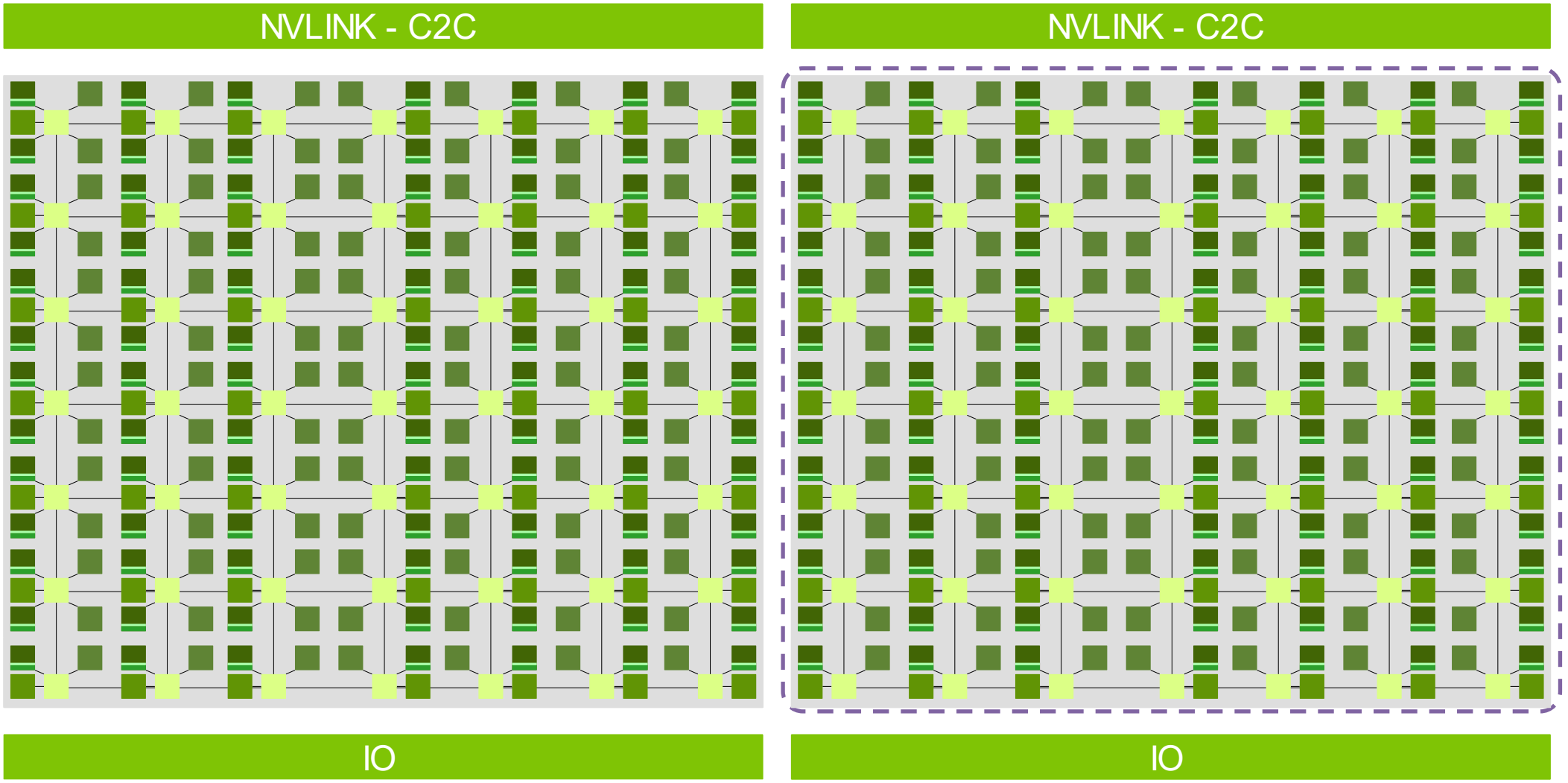
NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Vera/Olympus core architecture and AArch64 Arm ISA



Node architecture of Vera



1 ccNUMA domain

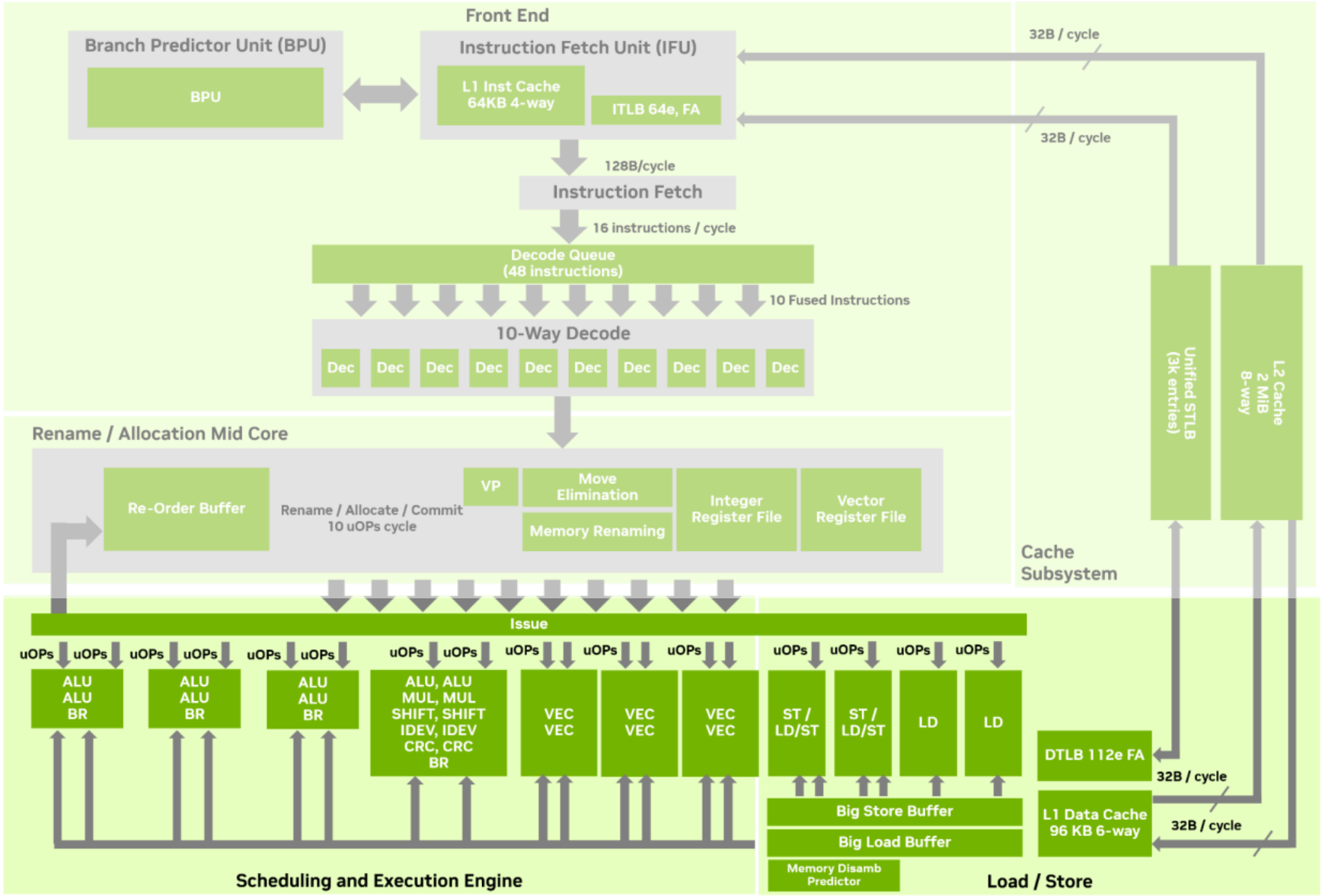
88 cores per socket

2 sockets per node

Port model for the Vera/Olympus

Frontend

Backend



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case Study: Maximize GEMM SIMD perf on Vera

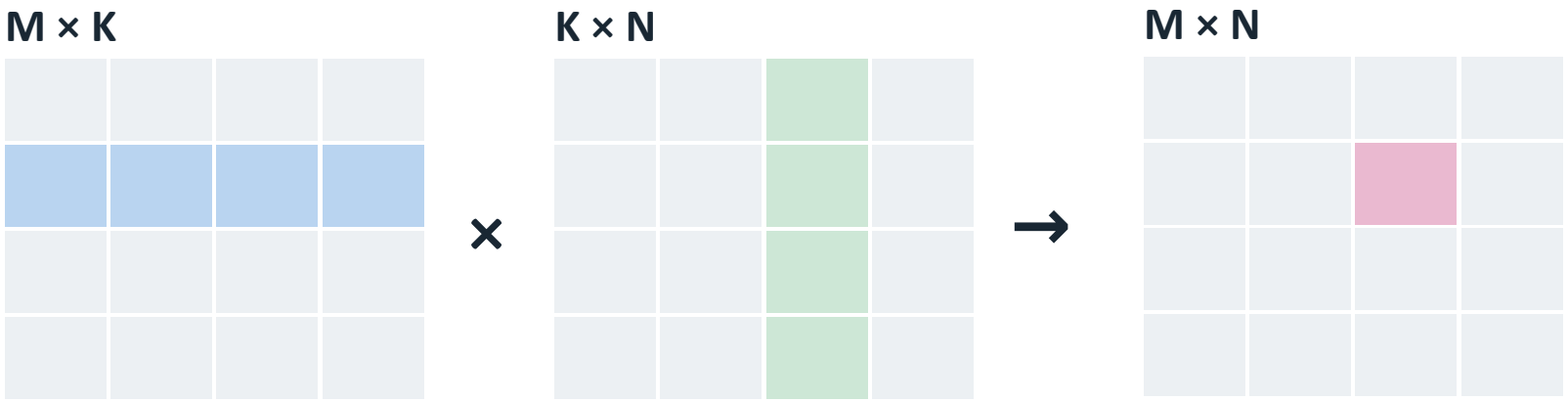


Context

- Matrix-Matrix multiplications (*GEMM) underpin many numerical simulations, scientific data analysis, and machine learning algorithms.
- GEMM operations offer high data reuse and substantial parallelism, making them well suited to SIMD execution, cache blocking, and tiling.
- Many implementations, majority hand-tuned assembly kernels

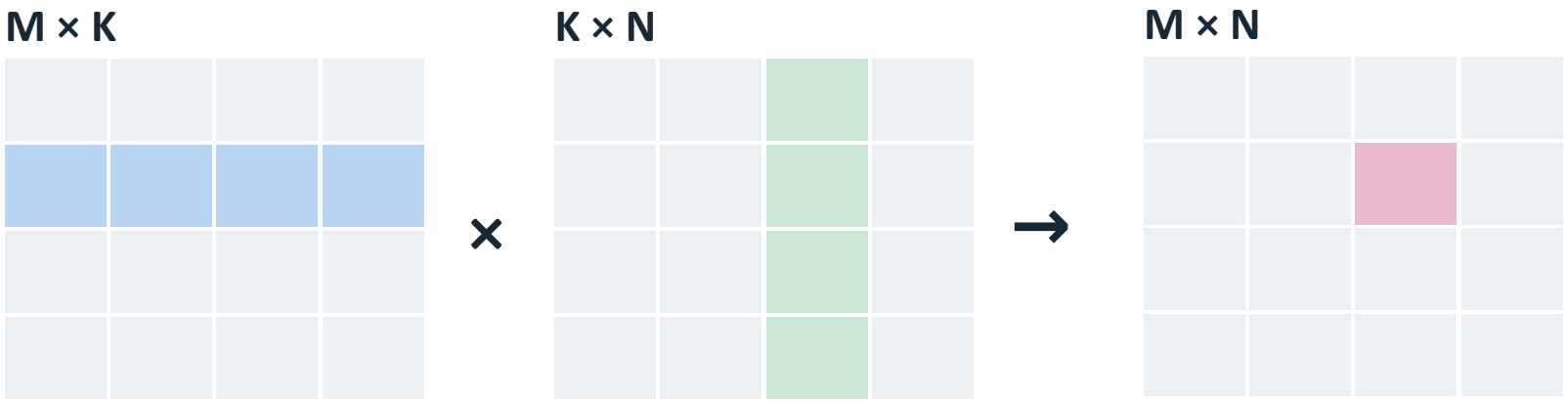
$$C := \alpha \cdot \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta \cdot C$$

GEMM Soul

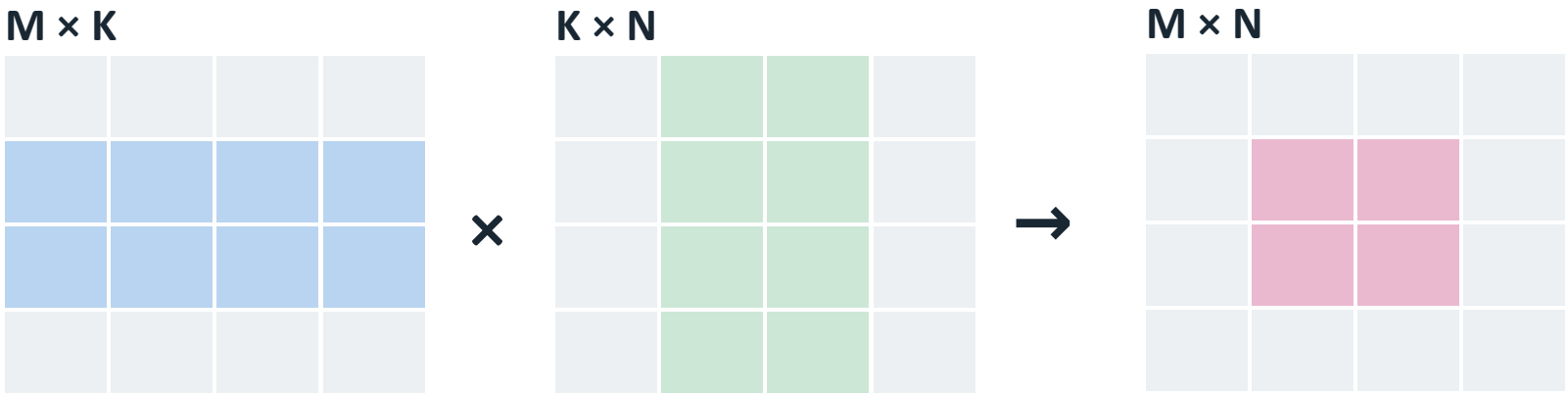


C[i,j] combines row i of A (all scalars) with column j of B (all scalars)

GEMM Soul

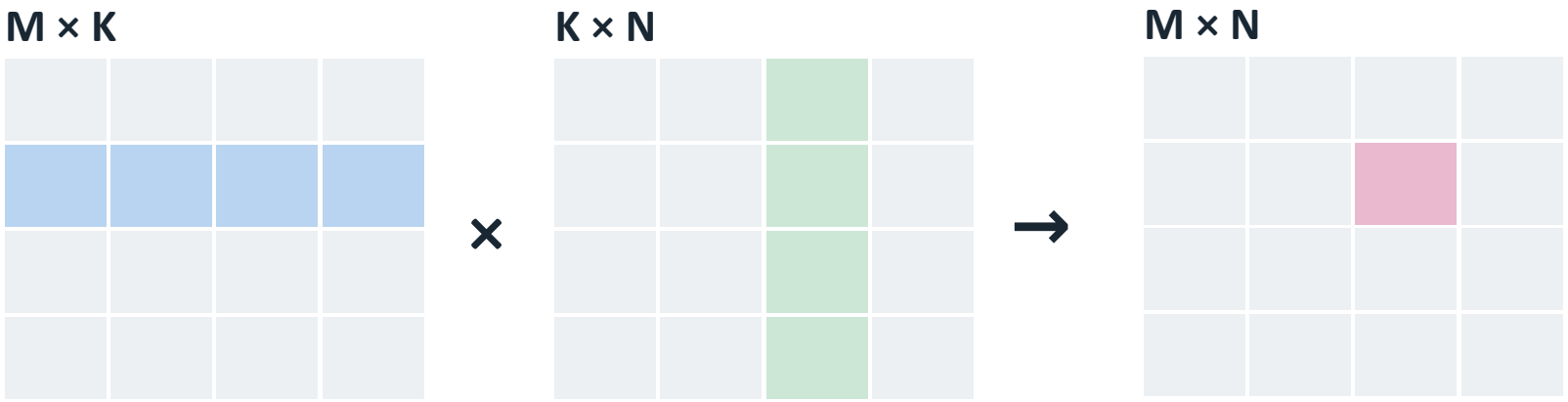


C[i,j] combines row i of A (all scalars) with column j of B (all scalars)

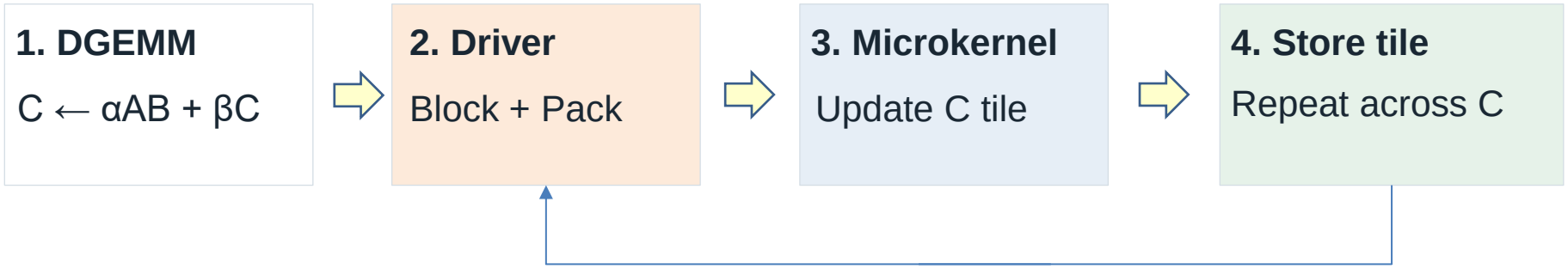


Compute per **tile** capable of maximise core utilization and resources

GEMM Soul



C[i,j] combines row i of A (all scalars) with column j of B (all scalars)



NOTE: Packed panels use cache space and generate traffic in L2 and L1 however is worth the effort to maximize compute

Area of Focus

- Popular GEMM (BLAS3) implementation: OpenBLAS
  - Follow the “tile” approach, similar to BLISS
  - Has specialized assembly kernel for GEMM
- Grace core (Neoverse V2) supported. Olympus not supported.
  - How well NEOVERSEV2 perform on Grace? (baseline)
  - How well NEOVERSEV2 perform on Olympus?
  - What is the “Speed-of-Light” on Olympus?

Area of Focus

- Popular GEMM (BLAS3) implementation: OpenBLAS
  - Follow the “tile” approach, similar to BLISS
  - Has specialized assembly kernel for GEMM
- Grace core (Neoverse V2) supported. Olympus not supported.
  - How well NEOVERSEV2 perform on Grace? (baseline)
  - How well NEOVERSEV2 perform on Olympus?
  - What is the “Speed-of-Light” on Olympus?

miniKernelGEMM: standalone proxy for the extracted packed DGEMM assembly

- A Reuse any specialized GEMM tile kernels form OpenBLAS
- Benchmark reuse packaged buffers multiple times, timing and profiling focused on tile computation

“Speed-of-Light” vs Measured SIMD Throughput

Neoverse V2 model
4 pipes × 2 FP64 lanes × 2 FLOPs
= 16 FLOPs / cycle

(48 GFLOP/s @ 3 GHz)

Olympus model
6 pipes × 2 FP64 lanes × 2 FLOPs
= 24 FLOPs / cycle

(72 GFLOP/s @ 3 GHz)

“Speed-of-Light” vs Measured SIMD Throughput

Neoverse V2 model
4 pipes × 2 FP64 lanes × 2 FLOPs
= 16 FLOPs / cycle

(48 GFLOP/s @ 3 GHz)

Olympus model
6 pipes × 2 FP64 lanes × 2 FLOPs
= 24 FLOPs / cycle

(72 GFLOP/s @ 3 GHz)

Platform	Kernel	IPC*	FLOPs / cycle*	Theoretical Throughput	SIMD Proxy Utilization*
Grace	NEOVERSEV2	4.51	13.03	16 ops/cycle	81.4%
Vera	NEOVERSEV2	5.43	15.69	24 ops/cycle	**66.5%**

(* measured via perf running 128x128 DGEMM problem which fit L1 on both microarchitectures)

FP64 ops/cycle = (FP_DP_FIXED_OPS_SPEC + FP_DP_SCALE_OPS_SPEC) ÷ CPU_CYCLES

SIMD proxy (%) = 100 × FP64 ops/cycle ÷ nominal FP64 ops/core/cycle

NEOVERSEV2 kernel (8×4)

Each cell = 128-bit accumulator holding two adjacent rows of C

	v8/v12	v9/v13	v10/v14	v11/v15
r0: v0-v1	**v16**	**v20**	**v24**	**v28**
r1: v2-v3	**v17**	**v21**	**v25**	**v29**
r2: v4-v5	**v18**	**v22**	**v26**	**v30**
r3: v6-v7	**v19**	**v23**	**v27**	**v31**

NEOVERSEV2 kernel (8×4)

Each cell = 128-bit accumulator holding two adjacent rows of C

	v8/v12	v9/v13	v10/v14	v11/v15
r0: v0-v1	**v16**	**v20**	**v24**	**v28**
r1: v2-v3	**v17**	**v21**	**v25**	**v29**
r2: v4-v5	**v18**	**v22**	**v26**	**v30**
r3: v6-v7	**v19**	**v23**	**v27**	**v31**

NEOVERSEV2 8 × 4 microkernel REGISTERS' ROLES

V0–V7 (A rows) | **V8–V15 · B values** | **V16–V31 · 16 independent output accumulators**

NEOVERSEV2 kernel (8×4)

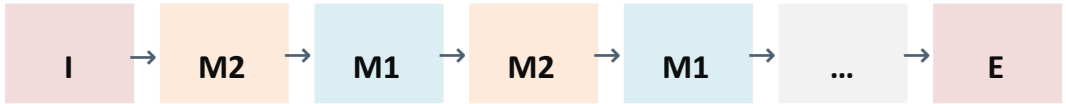
Each cell = 128-bit accumulator holding two adjacent rows of C

	v8/v12	v9/v13	v10/v14	v11/v15
r0: v0-v1	**v16**	**v20**	**v24**	**v28**
r1: v2-v3	**v17**	**v21**	**v25**	**v29**
r2: v4-v5	**v18**	**v22**	**v26**	**v30**
r3: v6-v7	**v19**	**v23**	**v27**	**v31**

NEOVERSEV2 8 × 4 microkernel REGISTERS' ROLES



MAIN LOOP:



M1 → A v0–v3, B v8–v11

M2 → A v4–v7, B v12–v15

NEOVERSEV2 kernel (M1)

blue: this stage | green: done | grey: pending

NEOVERSEV2 kernel (M1)

KERNEL8x4_M1 · v0–v3 + v8–v11

fmla v16.2d, v0.2d, v8.d[0]

fmla v20.2d, v0.2d, v9.d[0]

ldp q4, q5, [pA], #32

fmla v24.2d, v0.2d, v10.d[0]

fmla v28.2d, v0.2d, v11.d[0]

ldp d12, d13, [pB], #16

r0	**v16** #1	**v20** #2	**v24** #3	**v28** #4
r1	**v17** #5	**v21** #7	**v25** #6	**v29** #8
r2	**v18** #9	**v22** #10	**v26** #11	**v30** #12
r3	**v19** #13	**v23** #14	**v27** #15	**v31** #16

blue: this stage | green: done | grey: pending

NEOVERSEV2 kernel (M1)

KERNEL8x4_M1 · v0–v3 + v8–v11

fmla v16.2d, v0.2d, v8.d[0]

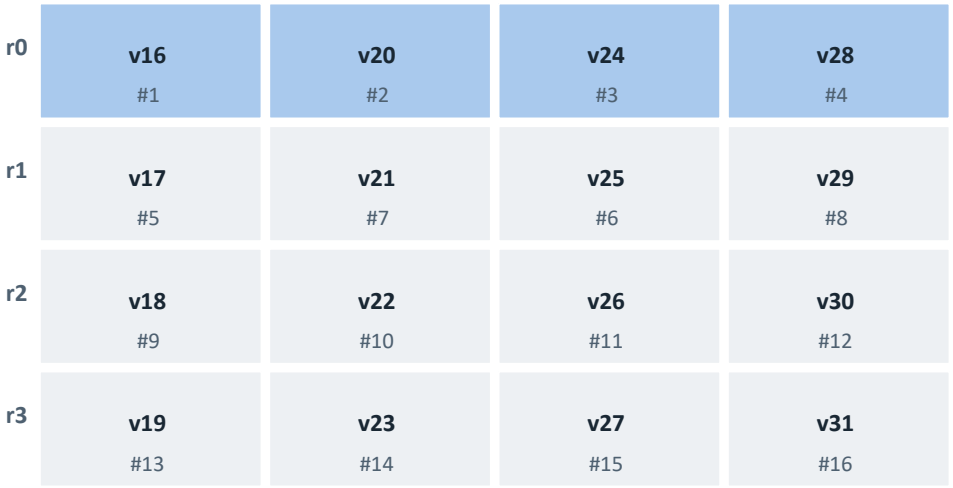
fmla v20.2d, v0.2d, v9.d[0]

ldp q4, q5, [pA], #32

fmla v24.2d, v0.2d, v10.d[0]

fmla v28.2d, v0.2d, v11.d[0]

ldp d12, d13, [pB], #16



KERNEL8x4_M1 · v0–v3 + v8–v11

fmla v17.2d, v1.2d, v8.d[0]

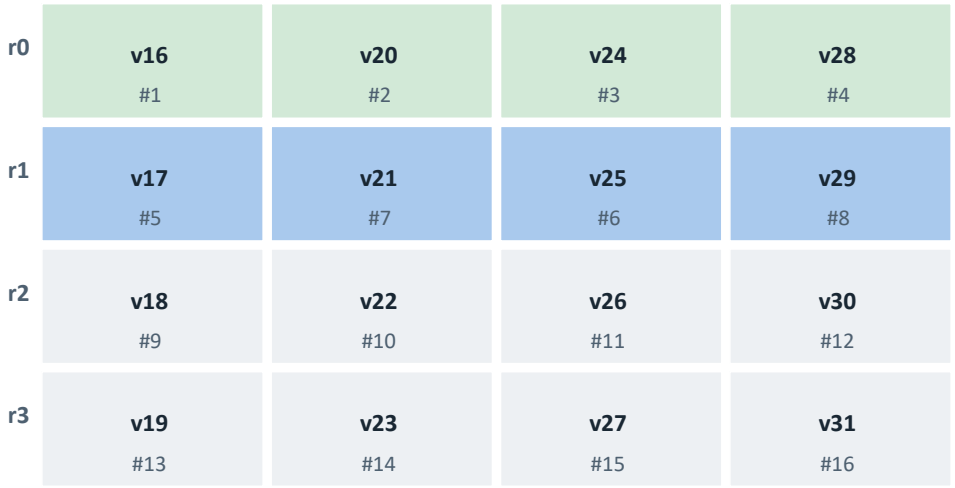
fmla v25.2d, v1.2d, v10.d[0]

prfm PLDL1KEEP, [pA, #A_PRE_SIZE+64]

fmla v21.2d, v1.2d, v9.d[0]

fmla v29.2d, v1.2d, v11.d[0]

ldp d14, d15, [pB], #16



blue: this stage | green: done | grey: pending

NEOVERSEV2 kernel (M1)

KERNEL8x4_M1 · v0–v3 + v8–v11

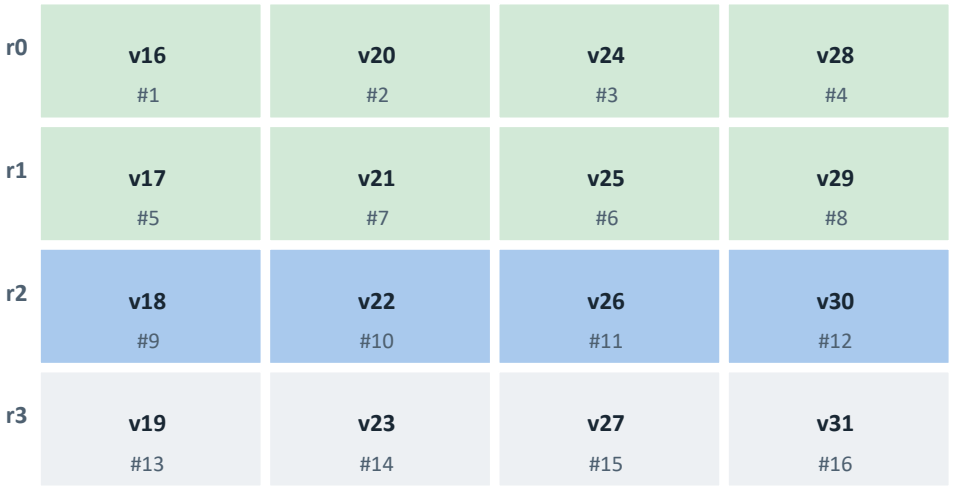
fmla v18.2d, v2.2d, v8.d[0]

fmla v22.2d, v2.2d, v9.d[0]

prfm PLDL1KEEP, [pA, #A_PRE_SIZE]

fmla v26.2d, v2.2d, v10.d[0]

fmla v30.2d, v2.2d, v11.d[0]



blue: this stage | green: done | grey: pending

NEOVERSEV2 kernel (M1)

KERNEL8x4_M1 · v0–v3 + v8–v11

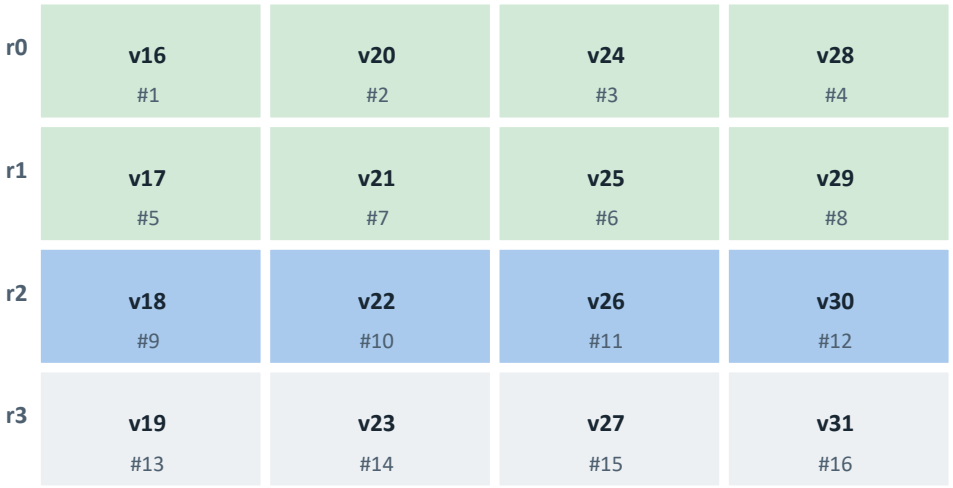
fmla v18.2d, v2.2d, v8.d[0]

fmla v22.2d, v2.2d, v9.d[0]

prfm PLDL1KEEP, [pA, #A_PRE_SIZE]

fmla v26.2d, v2.2d, v10.d[0]

fmla v30.2d, v2.2d, v11.d[0]



KERNEL8x4_M1 · v0–v3 + v8–v11

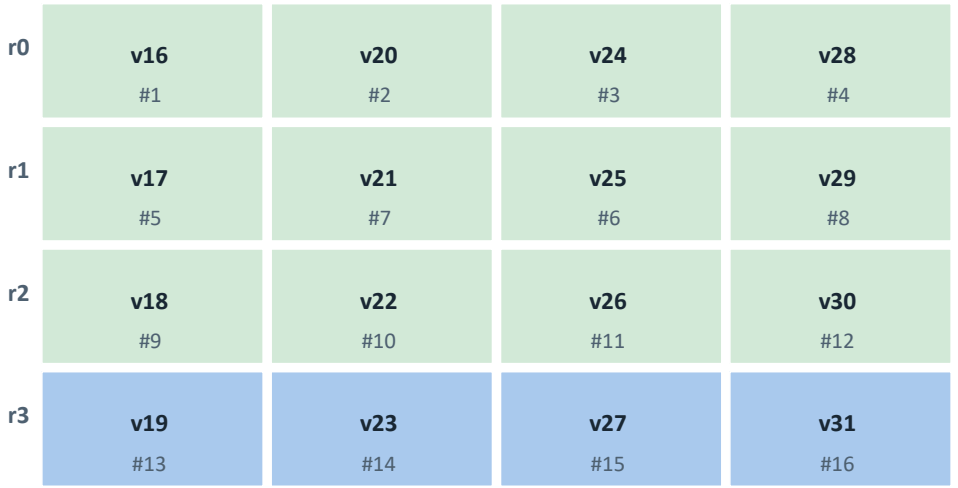
fmla v19.2d, v3.2d, v8.d[0]

fmla v23.2d, v3.2d, v9.d[0]

ldp q6, q7, [pA], #32

fmla v27.2d, v3.2d, v10.d[0]

1fmla v31.2d, v3.2d, v11.d[0]



blue: this stage | green: done | grey: pending

OLYMPUS kernel (12×4)

	v6[0] :B0	v6[1]: B1	v6[0]: B2	v6[1]: B3
v0: r0-1	**v16**	**v20**	**v24**	**v28**
v1: r2-3	**v17**	**v21**	**v25**	**v29**
v2: r4-5	**v18**	**v22**	**v26**	**v30**
v3: r6-7	**v19**	**v23**	**v27**	**v31**
v4: r8-9	**v18**	**v22**	**v26**	**v30**
v5: r10-11	**v19**	**v23**	**v27**	**v31**

OLYMPUS kernel (12×4)

	v6[0] :B0	v6[1]: B1	v6[0]: B2	v6[1]: B3
v0: r0-1	**v16**	**v20**	**v24**	**v28**
v1: r2-3	**v17**	**v21**	**v25**	**v29**
v2: r4-5	**v18**	**v22**	**v26**	**v30**
v3: r6-7	**v19**	**v23**	**v27**	**v31**
v4: r8-9	**v18**	**v22**	**v26**	**v30**
v5: r10-11	**v19**	**v23**	**v27**	**v31**

OLYMPUS 12 × 4 microkernel REGISTERS' ROLES

A · v0–v5 · 3 vectors / 12 rows | **B · v6–v7 · 4 columns** | **C · v8–v31 · 24 vectors / 48 outputs**

OLYMPUS kernel

blue: this stage | green: done | grey: pending

OLYMPUS kernel

FMLA ONLY

fmla v8.2d, v0.2d, v6.d[0]

fmla v15.2d, v1.2d, v6.d[1]

fmla v10.2d, v2.2d, v6.d[0]

fmla v14.2d, v0.2d, v6.d[1]

v0	**v8** #1	**v14** #4	**v20** #13	**v26** #14
v1	**v9** #5	**v15** #2	**v21** #15	**v27** #16
v2	**v10** #3	**v16** #6	**v22** #17	**v28** #18
v3	**v11** #10	**v17** #7	**v23** #19	**v29** #20
v4	**v12** #8	**v18** #11	**v24** #21	**v30** #22
v5	**v13** #12	**v19** #9	**v25** #23	**v31** #24

blue: this stage | green: done | grey: pending

OLYMPUS kernel

FMLA ONLY

fmla v8.2d, v0.2d, v6.d[0]

fmla v15.2d, v1.2d, v6.d[1]

fmla v10.2d, v2.2d, v6.d[0]

fmla v14.2d, v0.2d, v6.d[1]

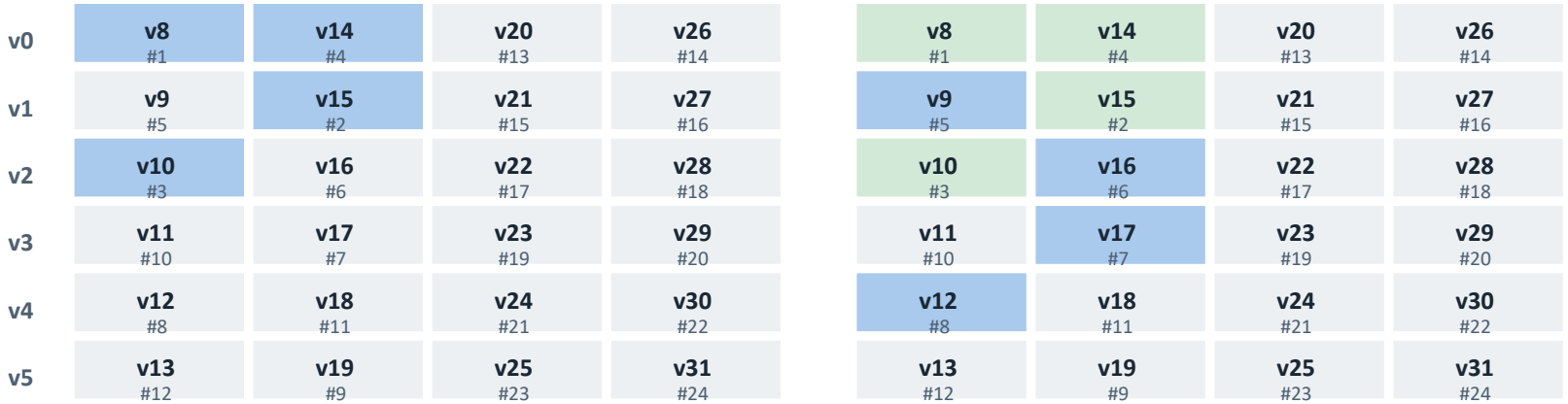
FMLA ONLY

fmla v9.2d, v1.2d, v6.d[0]

fmla v16.2d, v2.2d, v6.d[1]

fmla v17.2d, v3.2d, v6.d[1]

fmla v12.2d, v4.2d, v6.d[0]



blue: this stage | green: done | grey: pending

OLYMPUS kernel

FMLA ONLY

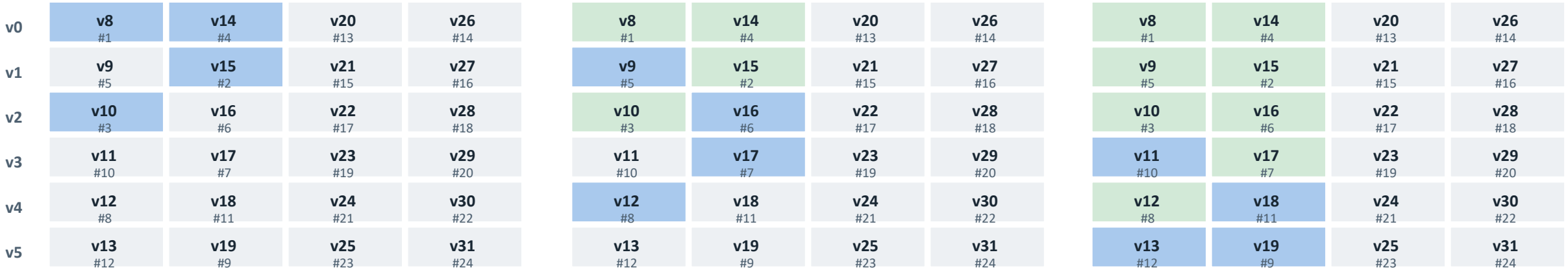
```
fmla v8.2d, v0.2d, v6.d[0]
fmla v15.2d, v1.2d, v6.d[1]
fmla v10.2d, v2.2d, v6.d[0]
fmla v14.2d, v0.2d, v6.d[1]
```

FMLA ONLY

```
fmla v9.2d, v1.2d, v6.d[0]
fmla v16.2d, v2.2d, v6.d[1]
fmla v17.2d, v3.2d, v6.d[1]
fmla v12.2d, v4.2d, v6.d[0]
```

FMLA + REFILL

```
fmla v19.2d, v5.2d, v6.d[1]
fmla v11.2d, v3.2d, v6.d[0]
fmla v18.2d, v4.2d, v6.d[1]
fmla v13.2d, v5.2d, v6.d[0]
ldr q6, [pB, #\boff0]
```



blue: this stage | green: done | grey: pending

OLYMPUS kernel

FMLA + REFILL

```
fmla v20.2d, v0.2d, v7.d[0]
fmla v26.2d, v0.2d, v7.d[1]
ldr q0, [pA, #\aoff0]
15  fmla v21.2d, v1.2d, v7.d[0]
16  fmla v27.2d, v1.2d, v7.d[1]
ldr q1, [pA, #\aoff1]
```

v0	**v8** #1	**v14** #4	**v20** #13	**v26** #14
v1	**v9** #5	**v15** #2	**v21** #15	**v27** #16
v2	**v10** #3	**v16** #6	**v22** #17	**v28** #18
v3	**v11** #10	**v17** #7	**v23** #19	**v29** #20
v4	**v12** #8	**v18** #11	**v24** #21	**v30** #22
v5	**v13** #12	**v19** #9	**v25** #23	**v31** #24

blue: this stage | green: done | grey: pending

OLYMPUS kernel

FMLA + REFILL

```
fmla v20.2d, v0.2d, v7.d[0]
fmla v26.2d, v0.2d, v7.d[1]
ldr q0, [pA, #\aoff0]
15  fmla v21.2d, v1.2d, v7.d[0]
16  fmla v27.2d, v1.2d, v7.d[1]
ldr q1, [pA, #\aoff1]
```

FMLA + REFILL

```
fmla v22.2d, v2.2d, v7.d[0]
fmla v28.2d, v2.2d, v7.d[1]
ldr q2, [x16, #\aoff0]
fmla v23.2d, v3.2d, v7.d[0]
fmla v29.2d, v3.2d, v7.d[1]
ldr q3, [x16, #\aoff1]
```

v0	v8 #1	v14 #4	v20 #13	v26 #14
v1	v9 #5	v15 #2	v21 #15	v27 #16
v2	v10 #3	v16 #6	v22 #17	v28 #18
v3	v11 #10	v17 #7	v23 #19	v29 #20
v4	v12 #8	v18 #11	v24 #21	v30 #22
v5	v13 #12	v19 #9	v25 #23	v31 #24

	v8 #1	v14 #4	v20 #13	v26 #14
	v9 #5	v15 #2	v21 #15	v27 #16
	v10 #3	v16 #6	v22 #17	v28 #18
	v11 #10	v17 #7	v23 #19	v29 #20
	v12 #8	v18 #11	v24 #21	v30 #22
	v13 #12	v19 #9	v25 #23	v31 #24

blue: this stage | green: done | grey: pending

OLYMPUS kernel

FMLA + REFILL

fmla v20.2d, v0.2d, v7.d[0]

fmla v26.2d, v0.2d, v7.d[1]

ldr q0, [pA, #\aoff0]

15 fmla v21.2d, v1.2d, v7.d[0]

16 fmla v27.2d, v1.2d, v7.d[1]

ldr q1, [pA, #\aoff1]

v0	v8 #1	v14 #4	v20 #13	v26 #14
v1	v9 #5	v15 #2	v21 #15	v27 #16
v2	v10 #3	v16 #6	v22 #17	v28 #18
v3	v11 #10	v17 #7	v23 #19	v29 #20
v4	v12 #8	v18 #11	v24 #21	v30 #22
v5	v13 #12	v19 #9	v25 #23	v31 #24

FMLA + REFILL

fmla v22.2d, v2.2d, v7.d[0]

fmla v28.2d, v2.2d, v7.d[1]

ldr q2, [x16, #\aoff0]

fmla v23.2d, v3.2d, v7.d[0]

fmla v29.2d, v3.2d, v7.d[1]

ldr q3, [x16, #\aoff1]

v8 #1	v14 #4	v20 #13	v26 #14
v9 #5	v15 #2	v21 #15	v27 #16
v10 #3	v16 #6	v22 #17	v28 #18
v11 #10	v17 #7	v23 #19	v29 #20
v12 #8	v18 #11	v24 #21	v30 #22
v13 #12	v19 #9	v25 #23	v31 #24

FMLA + REFILL

fmla v24.2d, v4.2d, v7.d[0]

fmla v30.2d, v4.2d, v7.d[1]

ldr q4, [x17, #\aoff0]

fmla v25.2d, v5.2d, v7.d[0]

fmla v31.2d, v5.2d, v7.d[1]

ldr q5, [x17, #\aoff1]

ldr q7, [pB, #\boff1]

v8 #1	v14 #4	v20 #13	v26 #14
v9 #5	v15 #2	v21 #15	v27 #16
v10 #3	v16 #6	v22 #17	v28 #18
v11 #10	v17 #7	v23 #19	v29 #20
v12 #8	v18 #11	v24 #21	v30 #22
v13 #12	v19 #9	v25 #23	v31 #24

blue: this stage | green: done | grey: pending

Results

Platform	Kernel	IPC*	FLOPs / cycle*	Theoretical Throughput	SIMD Proxy Utilization*
Grace	NEOVERSEV2	4.51	13.03	16 ops/cycle	81.4%
Vera	NEOVERSEV2	5.43	15.69	24 ops/cycle	**66.5%**
Vera	OLYMPUS	7.53	21.75	24 ops/cycle	**90.6%**

(* measured via perf running 128x128 DGEMM problem which fit L1 on both microarchitectures)

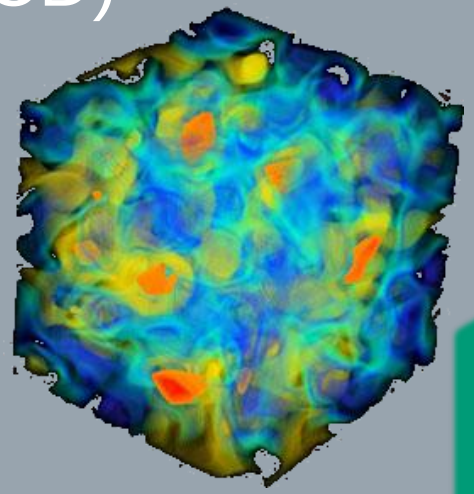
NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case Study: Domain Wall (DW) Kernel

from Quantum Chromodynamics (QCD)

on A64FX



© Brookhaven National Lab

Based on:

C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig:
ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.
Concurrency and Computation: Practice and Experience, e6512 (2021).

DOI: 10.1002/cpe.6512

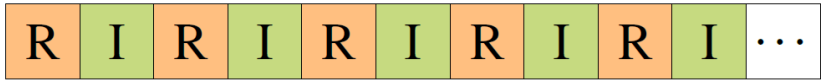
DW stencil kernel (simplified)

- “Grid” lattice QCD framework
- Uses SVE intrinsics
- Data type: double complex

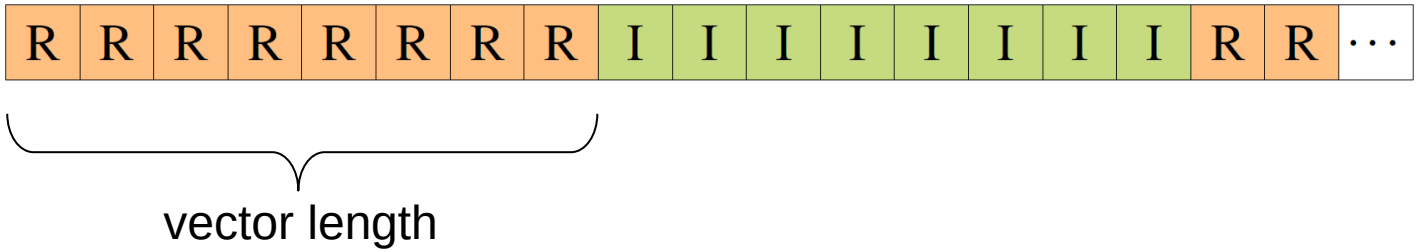
```
#define x_p 1 // x-plus direction
#define x_m 2 // x-minus direction
#define y_p 3 // y-plus direction
...
#pragma omp parallel for schedule(static)
for{t,z,y,x} = 1:{Lt-2,Lz-2,Ly-2,Lx-2} //collapsed loop over 4d space-time
{
  for(int s=0; s<Ls; ++s) //loop over fifth dimension
  {
    O[t][z][y][x][s] = R(x_p) · U[x_p][t][z][y][x] · P(x_p) · I[t][z][y][x+1][s] +
                       R(x_m) · U[x_m][t][z][y][x] · P(x_m) · I[t][z][y][x-1][s] +
                       R(y_p) · U[y_p][t][z][y][x] · P(y_p) · I[t][z][y+1][x][s] +
                       R(y_m) · U[y_m][t][z][y][x] · P(y_m) · I[t][z][y-1][x][s] +
                       R(z_p) · U[z_p][t][z][y][x] · P(z_p) · I[t][z+1][y][x][s] +
                       R(z_m) · U[z_m][t][z][y][x] · P(z_m) · I[t][z-1][y][x][s] +
                       R(t_p) · U[t_p][t][z][y][x] · P(t_p) · I[t+1][z][y][x][s] +
                       R(t_m) · U[t_m][t][z][y][x] · P(t_m) · I[t-1][z][y][x][s];
  }
}
```

Complex numbers data layout choice

AoS (standard)



AoSoA

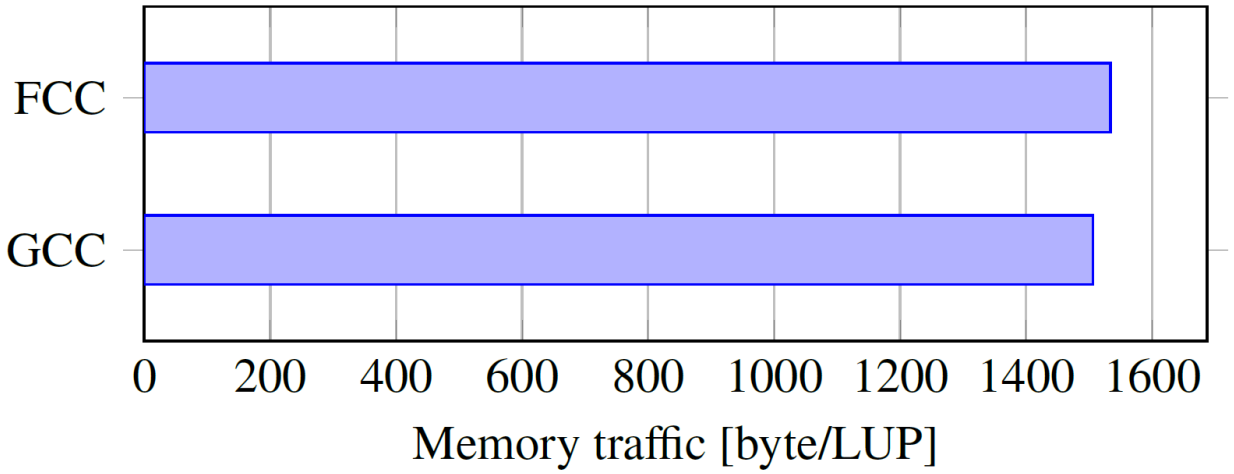


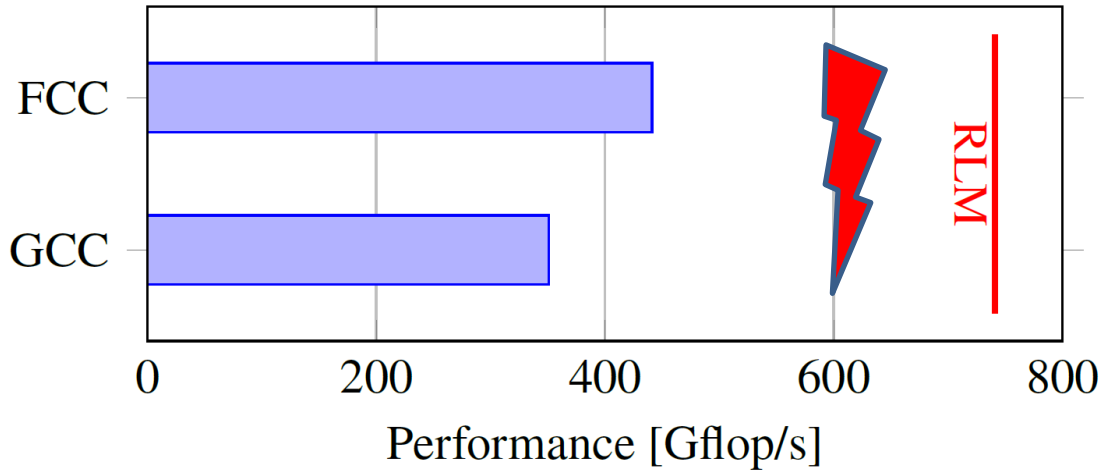
Observed performance

- Starting point: AoS layout, ACLE intrinsics, GCC/FCC
- 1320 flops/LUP (theoretical)
- Measured code balance: 1500 byte/LUP

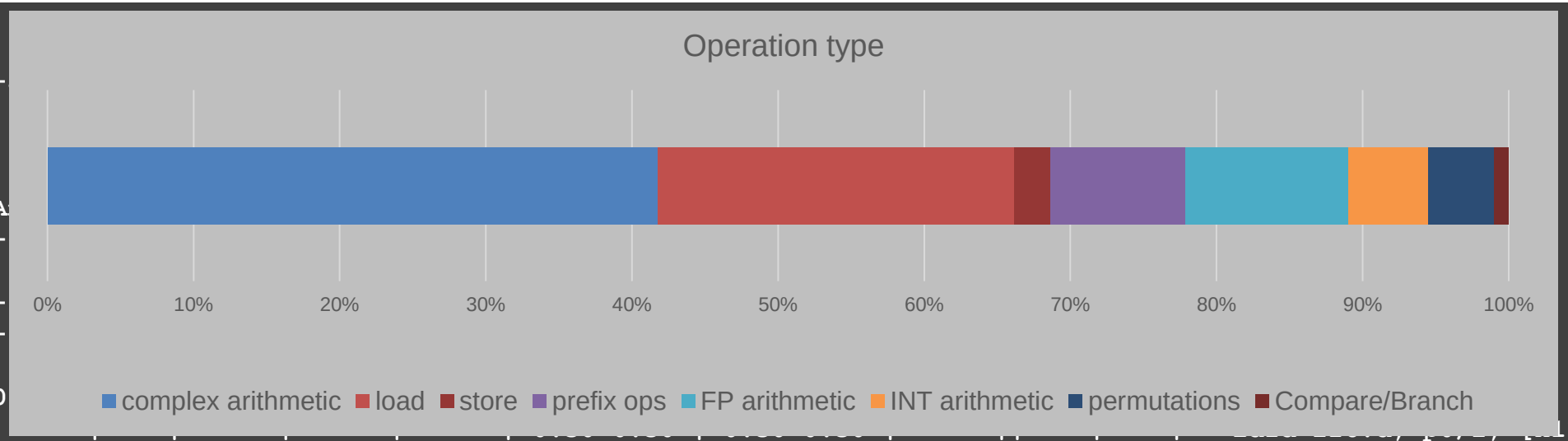
$$B_c \approx 1.14 \frac{\text{byte}}{\text{flop}}$$

- A64FX (FX1000): $B_m = 0.25 \frac{\text{byte}}{\text{flop}}$ → expect memory bound





In-core analysis (complex-AoS)




```
$ osaca --
[...]

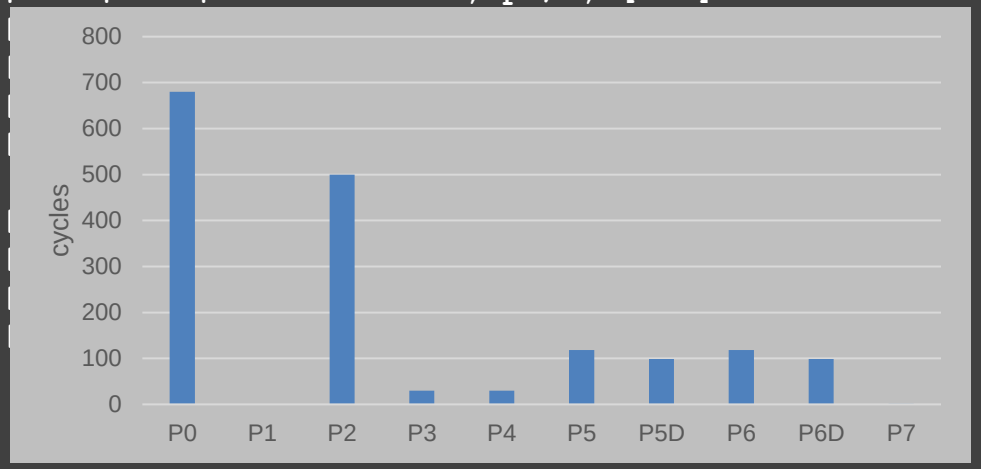
Combined A
----------

    | 0  -
----------
560 |
561 | 0.00
562 |
563 | 0.00     |    | 0.00 | 0.50 | 0.50 |
564 |          |    |      |      |      | 0.50 0.50 | 0.50 0.50 |
565 |          |    |      | 0.50 | 0.50 |
5
1
1
1369|          |    |      | 0.00 | 1.00 |
1370|          |    |      |      |      |           |           | 1.00 |
     680              500    30     30    118.5 98.5  118.5 98.5   1.0

Loop-Carried Dependencies Analysis Report
-----------------------------------------
1360 |  1.0 | add   w13, w13, 1                          | [1360]
```

FCMLA Zd, Pg, Zn, Zm, c **2cy on P0, 1cy on P2**

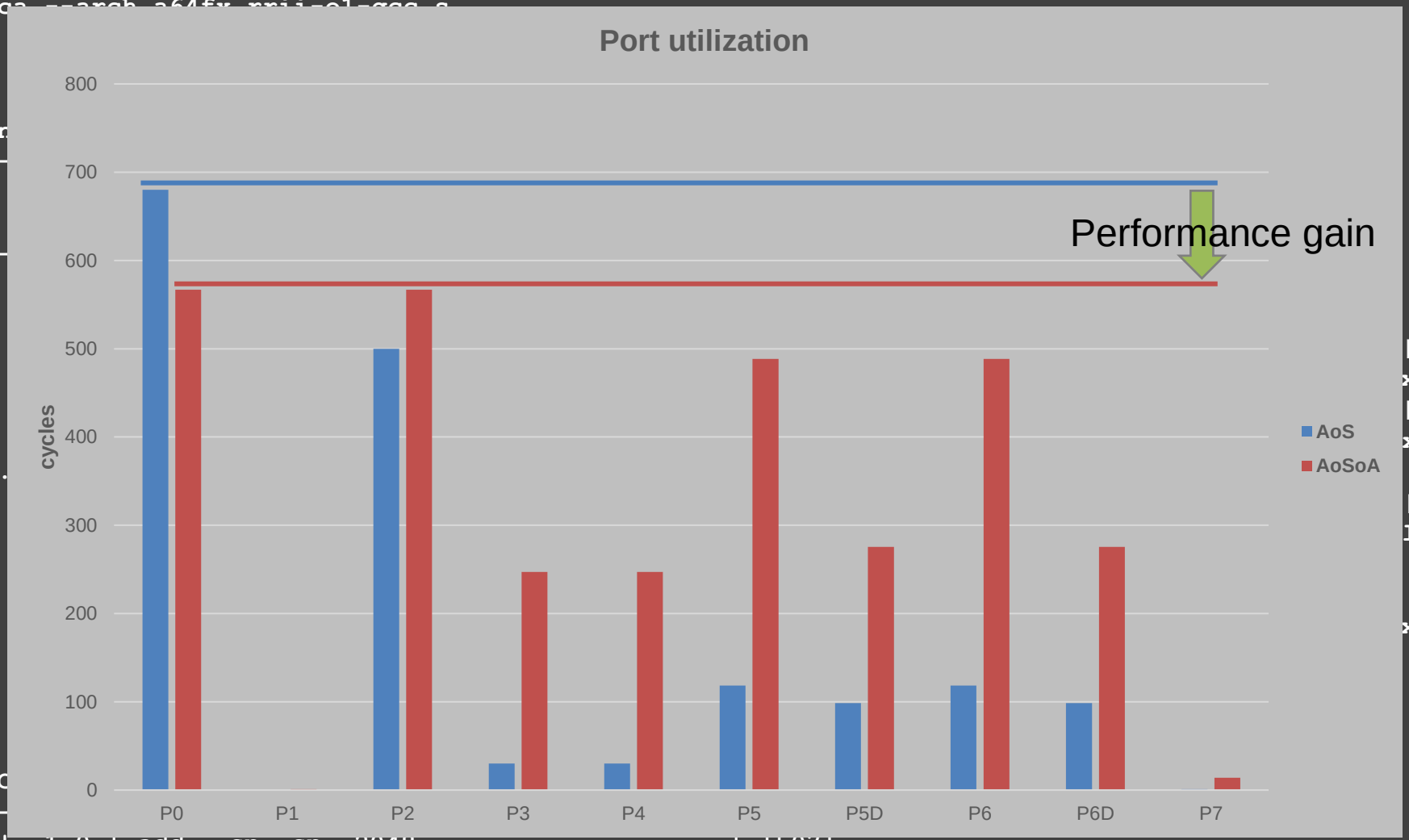
FCADD Zd, Pg, Zn, Zm, c **1cy on P0, 1cy on P2**

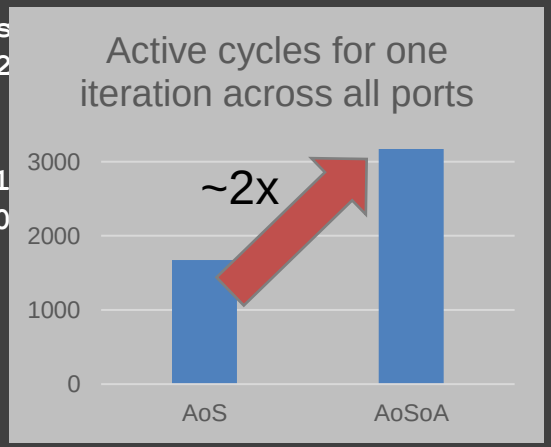


FMLA Zd, Pg, Zn, Zm **1cy on P0 OR P2**

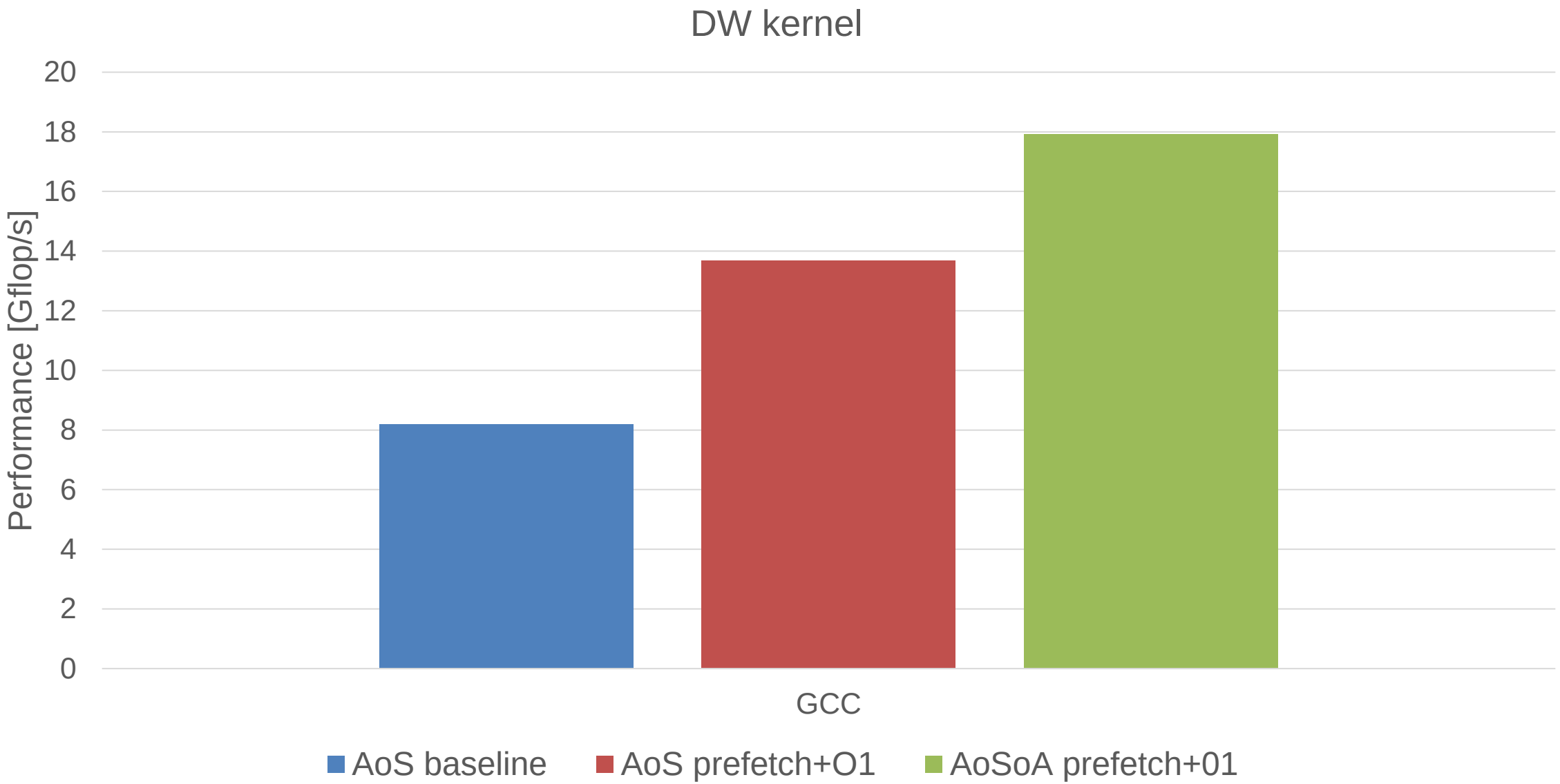
FADD Zd, Pg, Zn, Zm **1cy on P0 OR P2**

In-core analysis (complex-AoSoA)





DW kernel optimizations



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

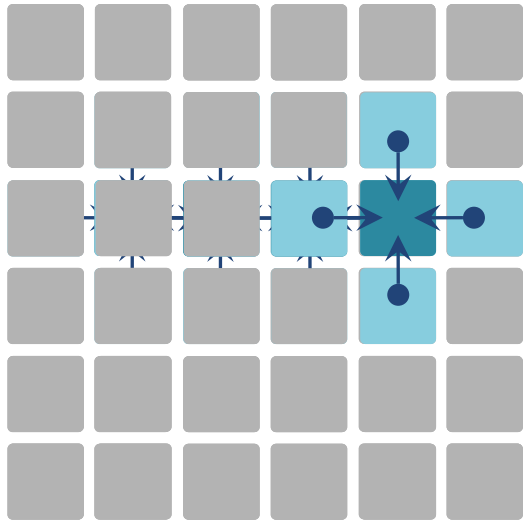
Hands-On #5: 2D Gauss-Seidel analysis

➔ https://go-nhr.de/CLPE-ex5



Hands-On: Gauss-Seidel Method

- Limited by loop-carried dependency
- Create code with **`-Ofast`**, **`-funroll-loops`**
- Analyze for SPR



```
for(int it=0; it<NITER; ++it) {
  for (int i=1; i<NI-1; ++i) {
    for (int k=1; k<NK-1; ++k) {
      phi[i][k] = 0.25 * (
          phi[i][k-1] + phi[i+1][k] +
          phi[i][k+1] + phi[i-1][k]
      );
    }
  }
}
```

→ **Moodle, hands-on #5**

Hands-On: Gauss-Seidel Method on SPR

```
Combined Analysis Report
------------------------

                                Port pressure in cycles
    | 0 - 0DV | 1 - 1DV |  2   |  3   |  4   |  5   |  6   |  7   |  8   |  9   |  10  |  11  ||  CP  | LCD  |
---------------------------------------------------------------------------------------------------------------
414 |         |         |      |      |      |      |      |      |      |      |      |      ||      |      | ..B1.72: # Preds ..B1.72 ..B1.71
415 |         |         | 0.00 | 0.00 |      |      |      |      |      |      | 1.00 |      || 5.0  |      | vmovsd 8(%rsi,%r10), %xmm2
416 | 0.00    |         |      |      |      | 0.00 | 1.00 |      |      |      |      | 0.00 ||      |      | incq %rdx
417 |         | 0.50    | 0.33 | 0.33 |      | 0.50 |      |      |      |      |      | 0.33 || 2.0  |      | vaddsd 16(%rsi,%r15), %xmm2, %xmm3
418 |         | 0.50    | 0.33 | 0.33 |      | 0.50 |      |      |      |      |      | 0.33 || 2.0  |      | vaddsd 8(%rsi,%r11), %xmm3, %xmm4
419 |         | 0.75    |      |      |      | 0.25 |      |      |      |      |      |      || 2.0  | 2.0  | vaddsd %xmm1, %xmm4, %xmm1
420 | 1.00    | 0.00    |      |      |      |      |      |      |      |      |      |      || 4.0  | 4.0  | vmulsd %xmm1, %xmm0, %xmm5
421 |         |         |      |      | 0.50 |      |      | 0.50 | 0.50 | 0.50 |      |      ||      |      | vmovsd %xmm5, 8(%rsi,%r15)
422 |         | 0.50    | 0.33 | 0.33 |      | 0.50 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 16(%rsi,%r10), %xmm5, %xmm6
423 |         | 0.50    | 0.33 | 0.33 |      | 0.50 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 24(%rsi,%r15), %xmm6, %xmm7
424 |         | 0.75    | 0.33 | 0.33 |      | 0.25 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 16(%rsi,%r11), %xmm7, %xmm8
425 | 1.00    | 0.00    |      |      |      |      |      |      |      |      |      |      || 4.0  | 4.0  | vmulsd %xmm8, %xmm0, %xmm9
426 |         |         |      |      | 0.50 |      |      | 0.50 | 0.50 | 0.50 |      |      ||      |      | vmovsd %xmm9, 16(%rsi,%r15)
427 |         | 0.50    | 0.33 | 0.33 |      | 0.50 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 24(%rsi,%r10), %xmm9, %xmm10
428 |         | 0.50    | 0.33 | 0.33 |      | 0.50 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 32(%rsi,%r15), %xmm10, %xmm11
429 |         | 0.75    | 0.33 | 0.33 |      | 0.25 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 24(%rsi,%r11), %xmm11, %xmm12
430 | 1.00    | 0.00    |      |      |      |      |      |      |      |      |      |      || 4.0  | 4.0  | vmulsd %xmm12, %xmm0, %xmm13
431 |         |         |      |      | 0.50 |      |      | 0.50 | 0.50 | 0.50 |      |      ||      |      | vmovsd %xmm13, 24(%rsi,%r15)
432 |         | 0.50    | 0.33 | 0.33 |      | 0.50 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 32(%rsi,%r10), %xmm13, %xmm14
433 |         | 0.25    | 0.33 | 0.33 |      | 0.75 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 40(%rsi,%r15), %xmm14, %xmm15
434 |         | 0.00    | 0.33 | 0.33 |      | 1.00 |      |      |      |      |      | 0.33 || 2.0  | 2.0  | vaddsd 32(%rsi,%r11), %xmm15, %xmm16
435 | 1.00    | 0.00    |      |      |      |      |      |      |      |      |      |      || 4.0  | 4.0  | vmulsd %xmm16, %xmm0, %xmm1
436 |         |         |      |      | 0.50 |      |      | 0.50 | 0.50 | 0.50 |      |      || 0.0  |      | vmovsd %xmm1, 32(%rsi,%r15)
437 | 0.00    | 0.00    |      |      |      |      | 0.50 |      |      |      | 0.50 |      ||      |      | addq $32, %rsi
438 | 0.00    |         |      |      |      | 0.00 | 0.50 |      |      |      | 0.50 |      ||      |      | cmpq %r13, %rdx
439 |         |         |      |      |      |      |      |      |      |      |      |      ||      |      | * jb ..B1.72 # Prob 28%

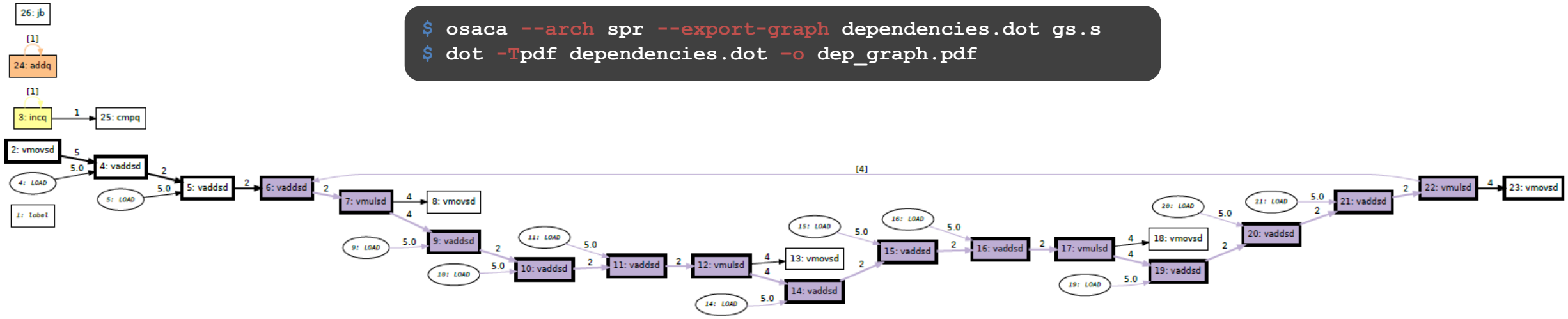
      4.00    6.00      3.67   3.67   2.00   6.00   2.00   2.00   2.00   2.00   2.00   3.67    45     36


Loop-Carried Dependencies Analysis Report
-----------------------------------------
416 |  1.0 | incq %rdx #143.11                        | [416]
419 | 36.0 | vaddsd %xmm1, %xmm4, %xmm1 #144.15       | [419, 420, 422, 423, 424, 425, 427, 428, 429, 430, 432, 433, 4
437 |  1.0 | addq $32, %rsi #143.11                   | [437]
```

Block Throughput	**1.50 cy**
Critical Path	**11.25 cy**
Loop-Carried Dep.	**9.0 cy**

Hands-On: Gauss-Seidel Method – standard version

```
$ osaca --arch spr --export-graph dependencies.dot gs.s
$ dot -Tpdf dependencies.dot -o dep_graph.pdf
```

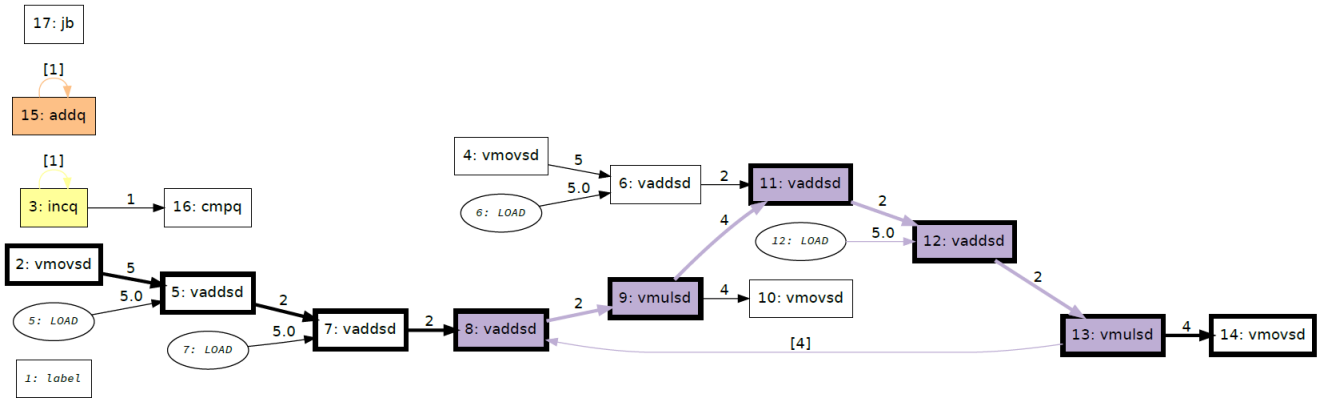


	LCD		
2		vmovsd	(%rsi,%r9), %xmm2
3		incq	%rdx
4		vaddsd	8(%rsi,%r10), %xmm2, %xmm3
5		vaddsd	16(%rsi,%r9), %xmm3, %xmm4
6	2.0	vaddsd	%xmm1, %xmm4, %xmm1
7	4.0	vmulsd	%xmm1, %xmm0, %xmm5
8		vmovsd	%xmm5, 8(%rsi,%r9)
9	2.0	vaddsd	(%rsi,%r10), %xmm5, %xmm6
10	2.0	vaddsd	8(%rsi,%r11), %xmm6, %xmm7
11	2.0	vaddsd	16(%rsi,%r10), %xmm7, %xmm8
12	4.0	vmulsd	%xmm8, %xmm0, %xmm9
13		vmovsd	%xmm9, 8(%rsi,%r10)
14	2.0	vaddsd	(%rsi,%r11), %xmm9, %xmm10
15	2.0	vaddsd	8(%rsi,%r8), %xmm10, %xmm11
16	2.0	vaddsd	16(%rsi,%r11), %xmm11, %xmm12
17	4.0	vmulsd	%xmm12, %xmm0, %xmm13
18		vmovsd	%xmm13, 8(%rsi,%r11)
19	2.0	vaddsd	(%rsi,%r8), %xmm13, %xmm14
20	2.0	vaddsd	8(%rsi,%r14), %xmm14, %xmm15
21	2.0	vaddsd	16(%rsi,%r8), %xmm15, %xmm16
22	4.0	vmulsd	%xmm16, %xmm0, %xmm1
23		vmovsd	%xmm1, 8(%rsi,%r8)
24		addq	%r13, %rsi
25		cmpq	%r12, %rdx
	36.0		

dep chain of 35 cy
7 out of 42 (CP) can overlap
➔ ratio 14.3%

Hands-On: Gauss-Seidel Method – opt. version (SPR)

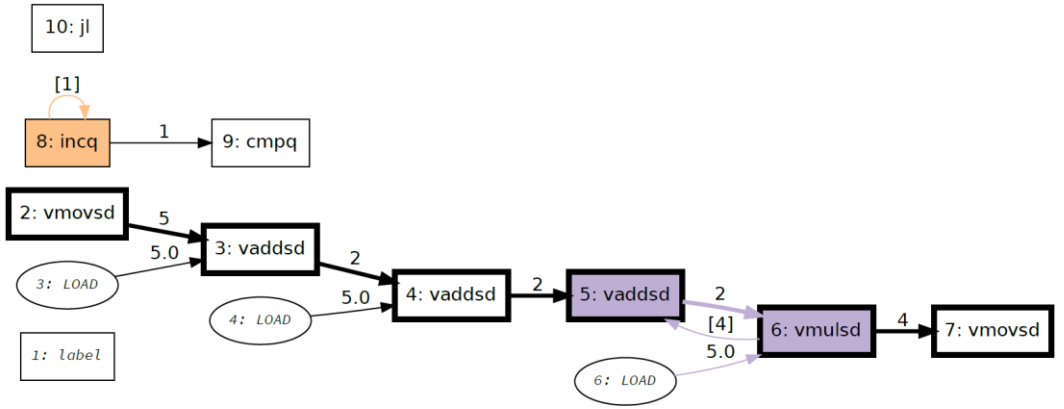
-Ofast / -O3




```
      LCD
  1 |       |    ..B1.72:
  2 |       |    vmovsd     8(%r10,%r11), %xmm2
  3 |       |    incq       %rdx
  4 |       |    vmovsd     16(%r10,%r11), %xmm5
  5 |       |    vaddsd     16(%r10,%rsi), %xmm2, %xmm3
  6 |       |    vaddsd     24(%r10,%rsi), %xmm5, %xmm7
  7 |       |    vaddsd     8(%r10,%r13), %xmm3, %xmm4
  8 | 2.0   |    vaddsd     %xmm1, %xmm4, %xmm1
  9 | 4.0   |    vmulsd     %xmm0, %xmm1, %xmm6
 10 |       |    vmovsd     %xmm6, 8(%r10,%rsi)
 11 | 2.0   |    vaddsd     %xmm7, %xmm6, %xmm8
 12 | 2.0   |    vaddsd     16(%r10,%r13), %xmm8, %xmm9
 13 | 4.0   |    vmulsd     %xmm0, %xmm9, %xmm1
 14 |       |    vmovsd     %xmm1, 16(%r10,%rsi)
 15 |       |    addq       $16, %r10
 16 |       |    cmpq       %r15, %rdx
 17 |       | * jb          ..B1.72
     14.0
```

dep chain of 14 cy
9cy / 23cy CP ➔ 39% overlap

-O1




```
      LCD
  1 |       |    ..B1.34:
  2 |       |    vmovsd     (%rsi,%rdi,8), %xmm0
  3 |       |    vaddsd     8(%rcx,%rdi,8), %xmm0, %xmm1
  4 |       |    vaddsd     (%rax,%rdi,8), %xmm1, %xmm2
  5 | 2.0   |    vaddsd     %xmm3, %xmm2, %xmm3
  6 | 4.0   |    vmulsd     .L_2il0floatpacket.0(%rip), %xmm3, %xmm3
  7 |       |    vmovsd     %xmm3, (%rcx,%rdi,8)
  8 |       |    incq       %rdi
  9 |       |    cmpq       %r13, %rdi
 10 |       | * jl          ..B1.34
     8.0
```

dep chain of 6cy
9 cy / 15 cy CP ➔ 60% overlap

Hands-On: Gauss-Seidel Method on SPR

Prediction [cy/it]	standard			optimized		
	`-Ofast`	`-O3`	`-O1`	`-Ofast`	`-O3`	`-O1`
icc 2021	9	9	14	7	7	6
icx 2022	9	9	8	8	8	8
icx 2024	9	11	10	8	10	10
GCC 14.2	8	10	10.5	8	4	6
Clang 18	10	10	10	10	4	4

Summary & Caveats

- A code analyzer helps you to predict the **in-core** runtime of a basic block
  - Might be sufficient, but often a full analysis requires a **memory model** as well!
- An analysis of **(loop-carried-)dependencies** can help you find performance limitations!
- Analysis is done on **compiler-generated** code which always holds a factor of uncertainty

- There might be additional things slowing you, e.g.:
  - Cache trashing
  - Loads across cache lines
  - Front end limitations
  - Bank conflicts
  - ...

There is not just THE one code analyzer

- **OSACA**: https://github.com/RRZE-HPC/OSACA
- **uiCA**: https://www.uops.info/uiCA.html
- **LLVM-MCA**: https://llvm.org/docs/CommandGuide/llvm-mca.html
- **IACA** (EoL): https://www.intel.com/content/www/us/en/developer/articles/tool/architecture-code-analyzer.html

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Thank you! Questions?

OSACA: https://github.com/RRZE-HPC/osaca

pip: `$ pip install -u osaca`

Compiler Explorer: https://godbolt.org

Survey: https://go-nhr.de/course-feedback