ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

LAB04: PULP-NN - Deep Neural Network Inference on PULP

Davide Nadalini – d.nadalini@unibo.it
Alberto Dequino – alberto.dequino@unibo.it
Francesco Conti – f.conti@unibo.it

Objective of the Class

Intro: PULP platform and the PULP-SDK

Tasks: some basics of C programming on PULP:

- Parallelization on the PULP architecture
- Matrix-multiplication
- Fully Connected layer with vectorized instructions
- 2Dconv
- profiling code execution

Deadline:

Oct 31st 2025

Programming Language: C

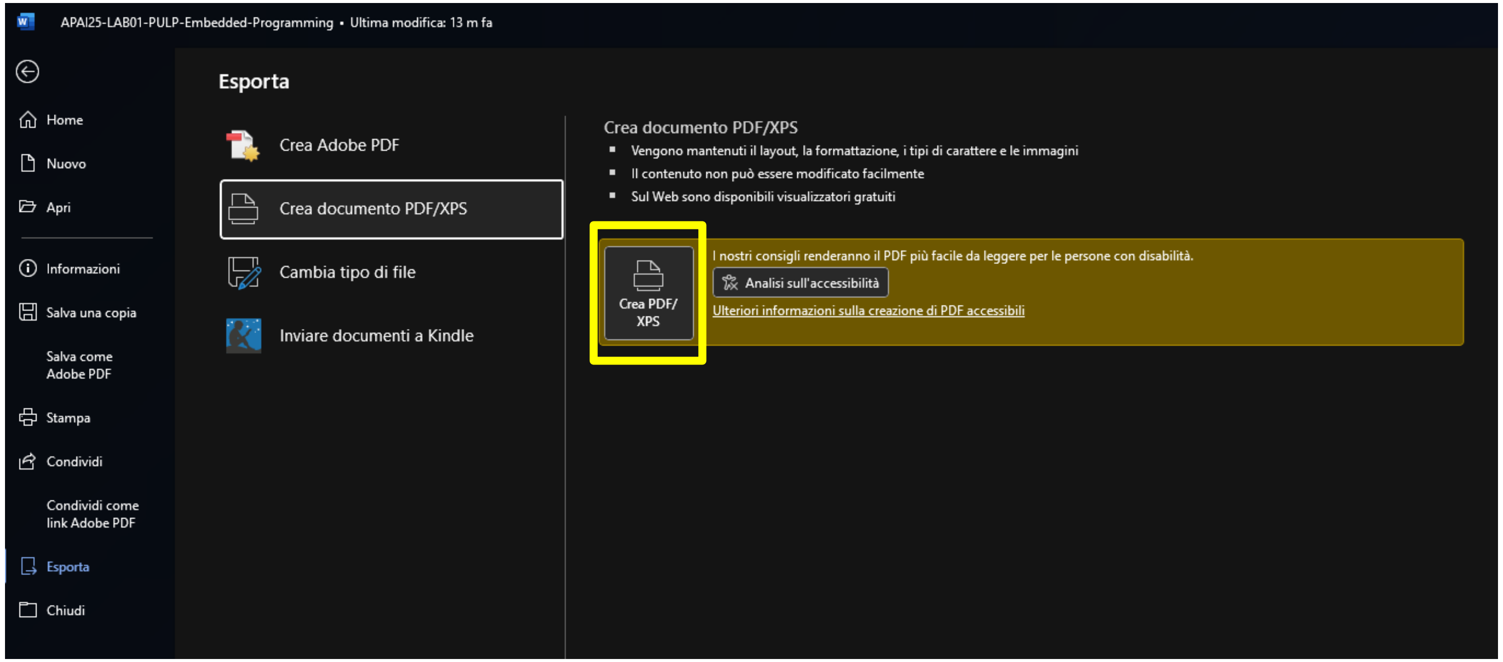
Lab duration: 3h

The class is meant to be interactive: coding together and on your own!

How to deliver the Assignment

You will deliver ONLY the PDF assignment, no code

- Download the assignment file from Virtuale.
- Fill the results required by the assignment.
- Export to pdf format.
- Rename the file to: LAB<number_of_the_lesson>_APAI_<your_name>.pdf
- Use Virtuale platform to load ONLY your .pdf file



APAI25-LAB01-PULP-Embedded-Programming • Ultima modifica: 13 m fa

Crea documento PDF/XPS

- Vengono mantenuti il layout, la formattazione, i tipi di carattere e le immagini
- Il contenuto non può essere modificato facilmente
- Sul Web sono disponibili visualizzatori gratuiti

I nostri consigli renderanno il PDF più facile da leggere per le persone con disabilità.

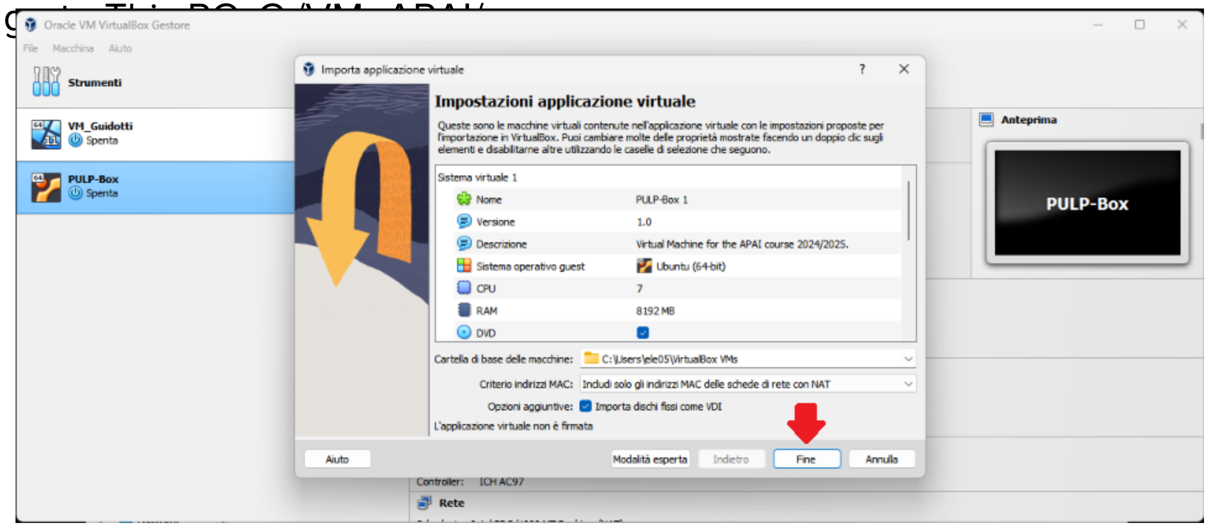
Opening the VM and VSCode

1. On the lab’s PCs, open the file explorer and g

2. Double click on PULP-box.ova

3. VirtualBox opens, just click on “Fine”

4. Wait for the VM to be imported



Impostazioni applicazione virtuale

Queste sono le macchine virtuali contenute nell'applicazione virtuale con le impostazioni proposte per l'importazione in VirtualBox. Puoi cambiare molte delle proprietà mostrate facendo un doppio clic sugli elementi e disabilitarne altre utilizzando le caselle di selezione che seguono.

Sistema virtuale 1

Nome	PULP-Box 1
Versione	1.0
Descrizione	Virtual Machine for the APAI course 2024/2025.
Sistema operativo guest	Ubuntu (64-bit)
CPU	7
RAM	8192 MB
DVD	☑

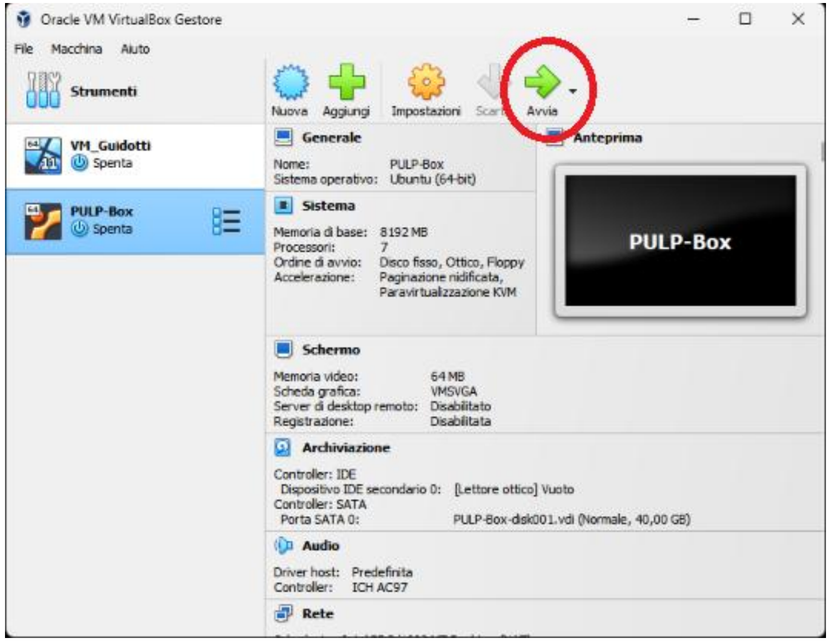
Cartella di base delle macchine: C:\Users\ele05\VirtualBox VMs

Criterio indirizzi MAC: Includi solo gli indirizzi MAC delle schede di rete con NAT

Opzioni aggiuntive: ☑ Importa dischi fissi come VDI

L'applicazione virtuale non è firmata

5. Open the VM with “Avvia”



Generale

Nome: PULP-Box
Sistema operativo: Ubuntu (64-bit)

Sistema

Memoria di base: 8192 MB
Processori: 7
Ordine di avvio: Disco fisso, Ottico, Floppy
Accelerazione: Paginazione nidificata, Paravirtualizzazione KVM

Schermo

Memoria video: 64 MB
Scheda grafica: VMSVGA
Server di desktop remoto: Disabilitato
Registrazione: Disabilitata

Archiviazione

Controller: IDE
Dispositivo IDE secondario 0: [Lettore ottico] Vuoto
Controller: SATA
Porta SATA 0: PULP-Box-disk001.vdi (Normale, 40,00 GB)

Audio

Driver host: Predefinita
Controller: ICH AC97

Rete

Password is ‘pulp’

Opening the Docker with VSCode

1. Open a terminal (right click – open a new terminal)
2. From the terminal, open VSCode in the folder of the Docker

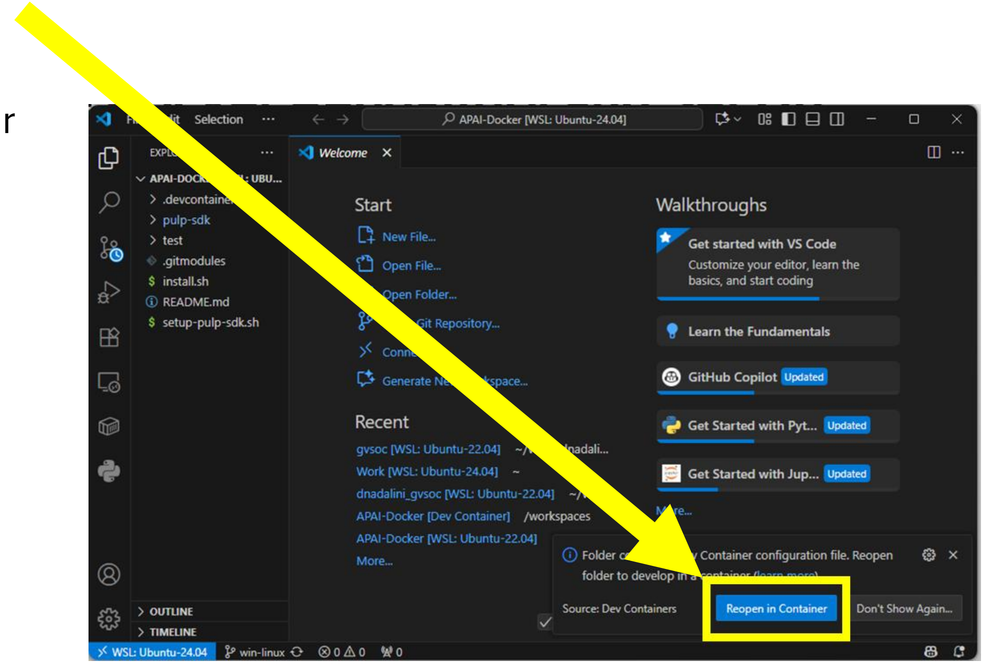
```
$ cd APAI-Docker
$ code .
```

1. Reopen the APAI-Docker folder in VSCode (click on “Reopen in container”)

1. Now you can use the integrated terminal (open with CTRL+J) to run your applications!

IMPORTANT: every time you open a new terminal to work on PULP, launch

```
$ source setup-pulp-sdk.sh
```



Folder c… …Container configuration file. Reopen folder to develop in a …

Source: Dev Containers

Getting Started:

IMPORTANT: activate the pulp-sdk module file <u>every</u> time a new shell is open.

```
$ source setup-pulp-sdk.sh
```

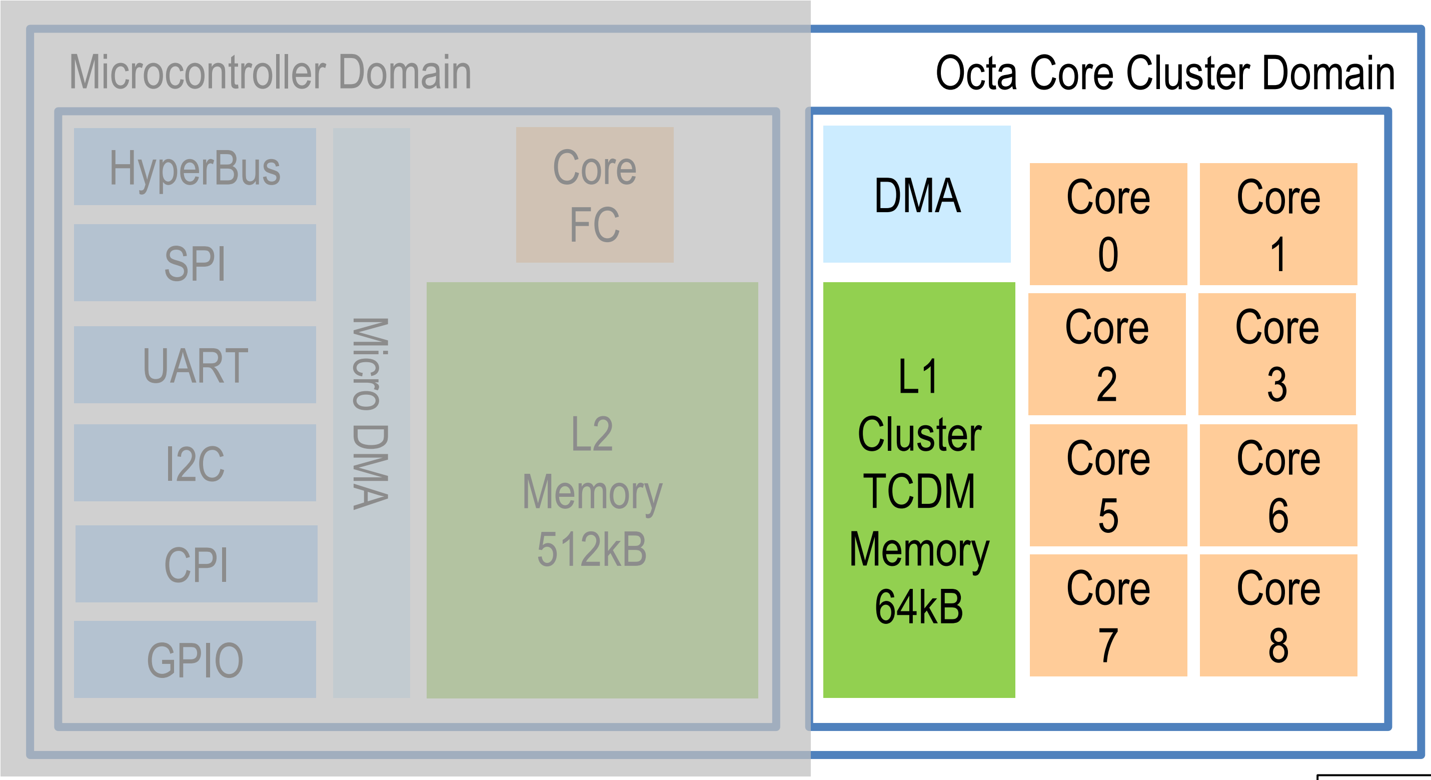
HOW TO RUN THE CODE:

```
$ git clone https://github.com/EEESlab/APAI25-LAB04-PULP-NN
$ cd APAI25-LAB04-PULP-NN
$ cd <folder_you_want>
$ make clean all run
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

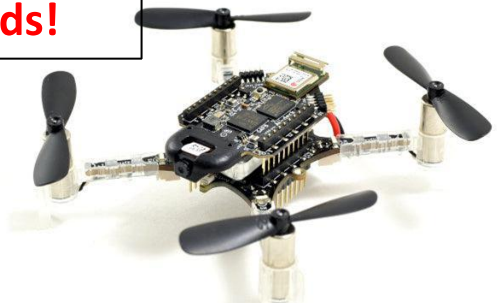
INTRO

PULP Platform: today we focus on the 8-cores cluster



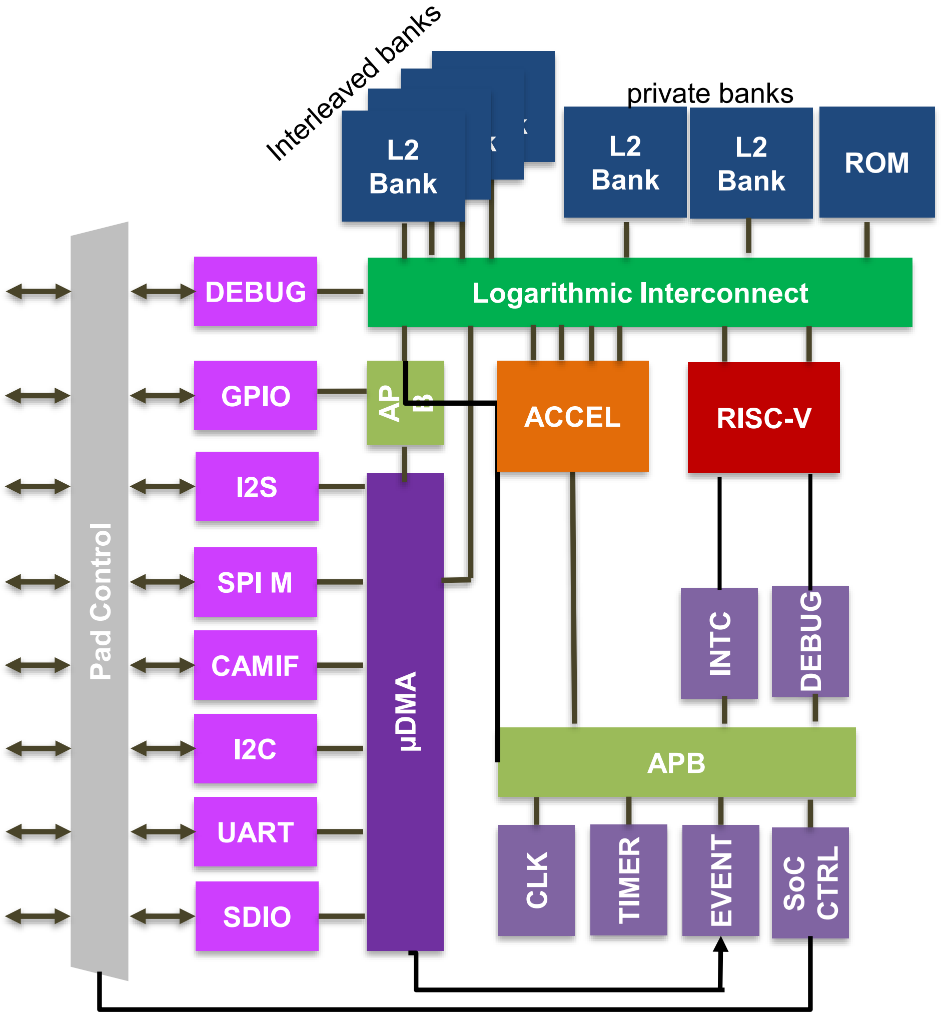
- **Cores**: 1 + 8
- **On-chip Memories**
  - A level 2 Memory, shared among all cores
  - A level 1 Memory, shared by the 8-cores cluster
- **cluster-DMA:** A multi-channel 1D/2D DMA, controlling the transactions between the L2 and L1 memories
- **micro-DMA**: A smart, lightweight and completely autonomous DMA () capable of handling complex I/O scheme
  - **Bus+Peripherals:** HyperBus, I2S, CPI, timers, SPI, GPIOs, etc…

NB: this is the architecture you find on our nano-drones and GAP boards!

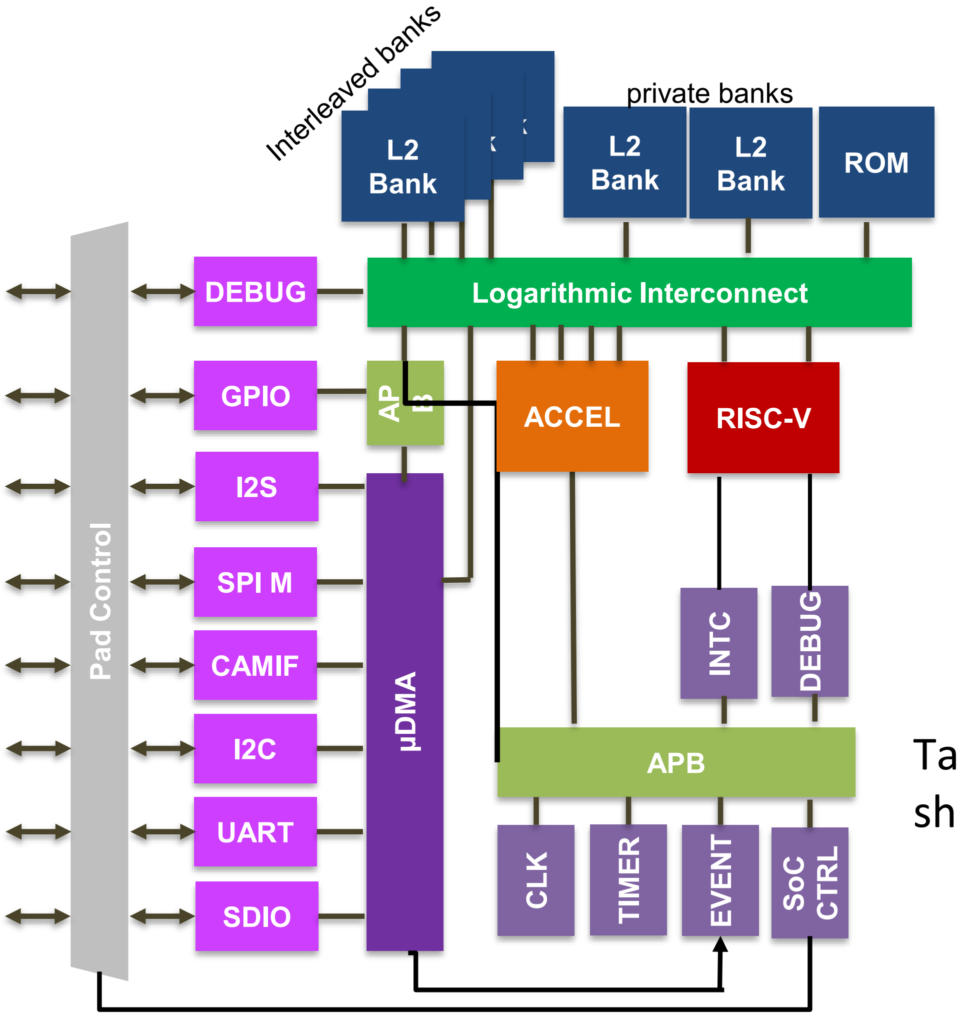


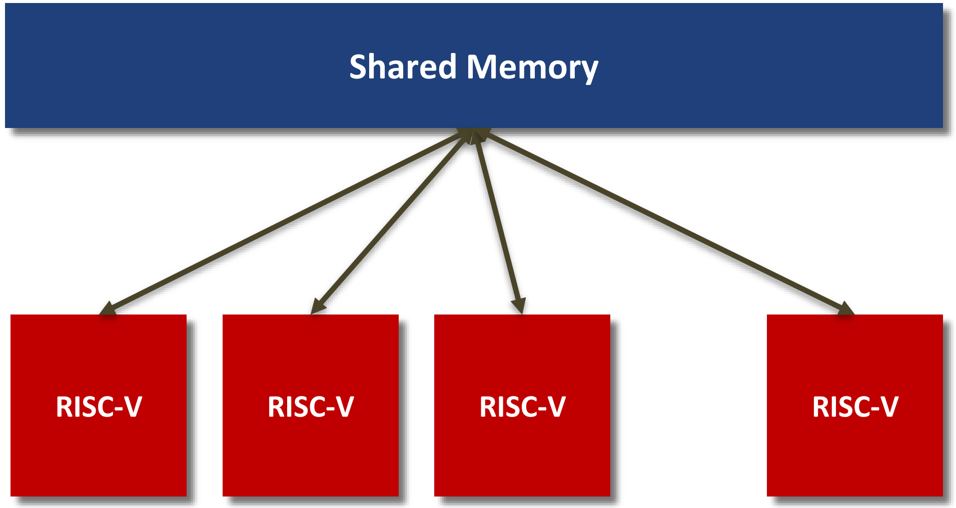
GitHub HW Project: https://github.com/pulp-platform/pulp
HW Documentation: https://raw.githubusercontent.com/pulp-platform/pulp/master/doc/datasheet.pdf

PULP: a Parallel Ultra-Low-Power computing platform



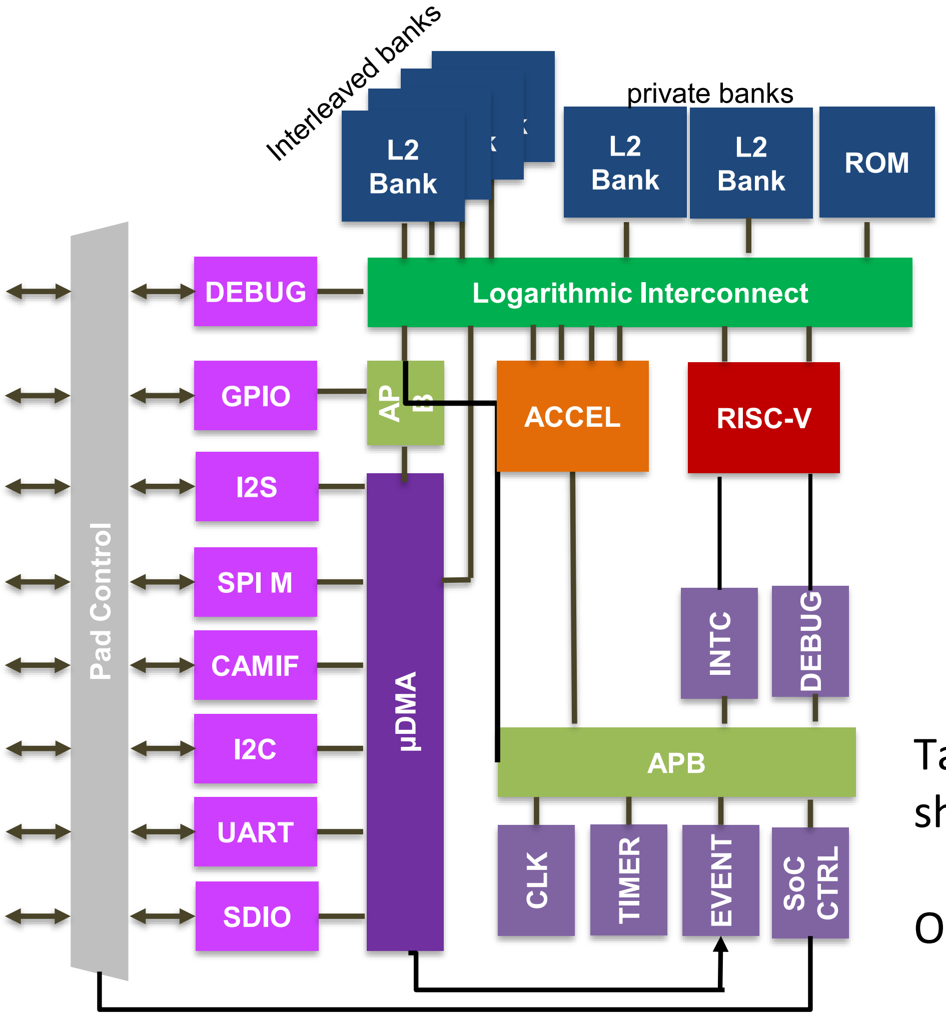
PULP: a Parallel Ultra-Low-Power computing platform

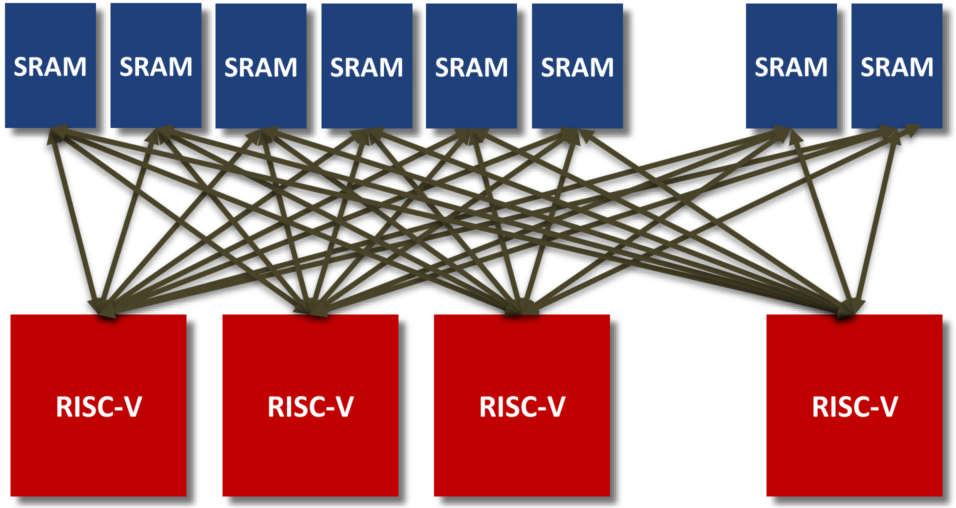




Target a **Shared-Memory** parallel programming model: 4—16 DSP cores sharing directly a **Shared L1 Memory** (***Tightly Coupled Data Mem*** or ***TCDM***)

PULP: a Parallel Ultra-Low-Power computing platform

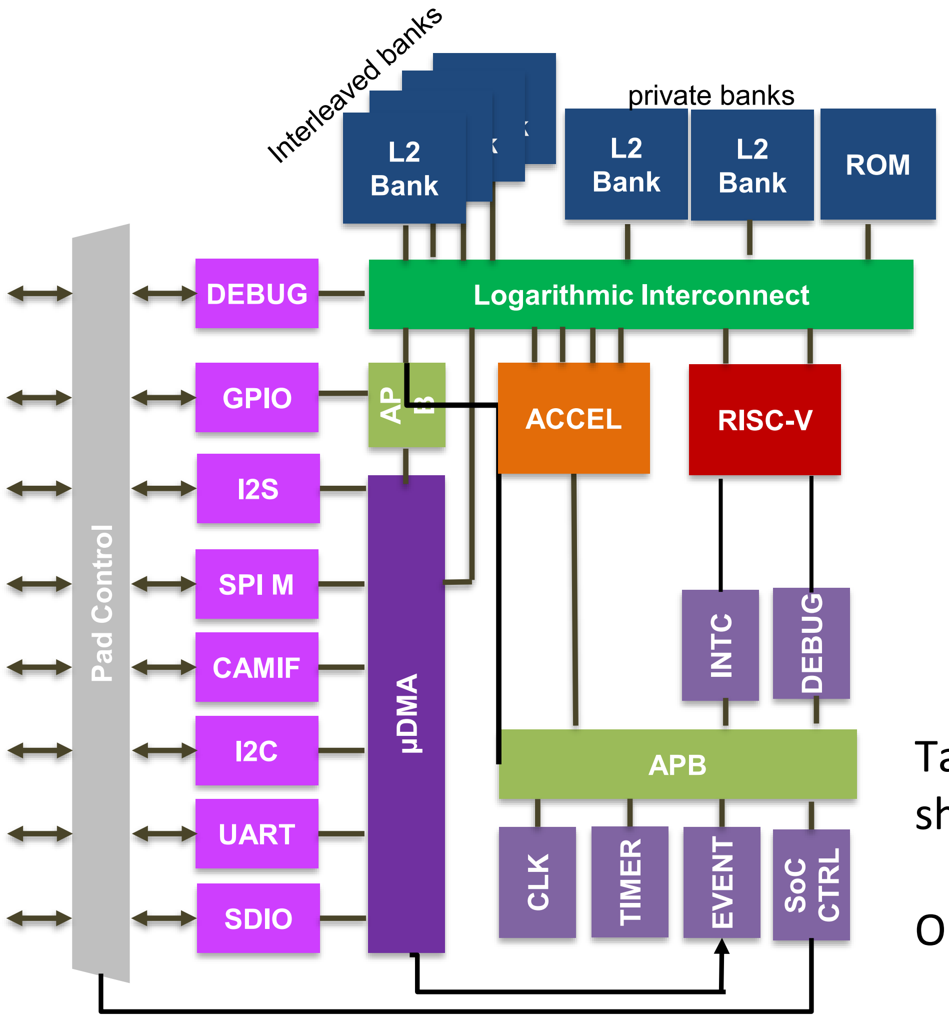


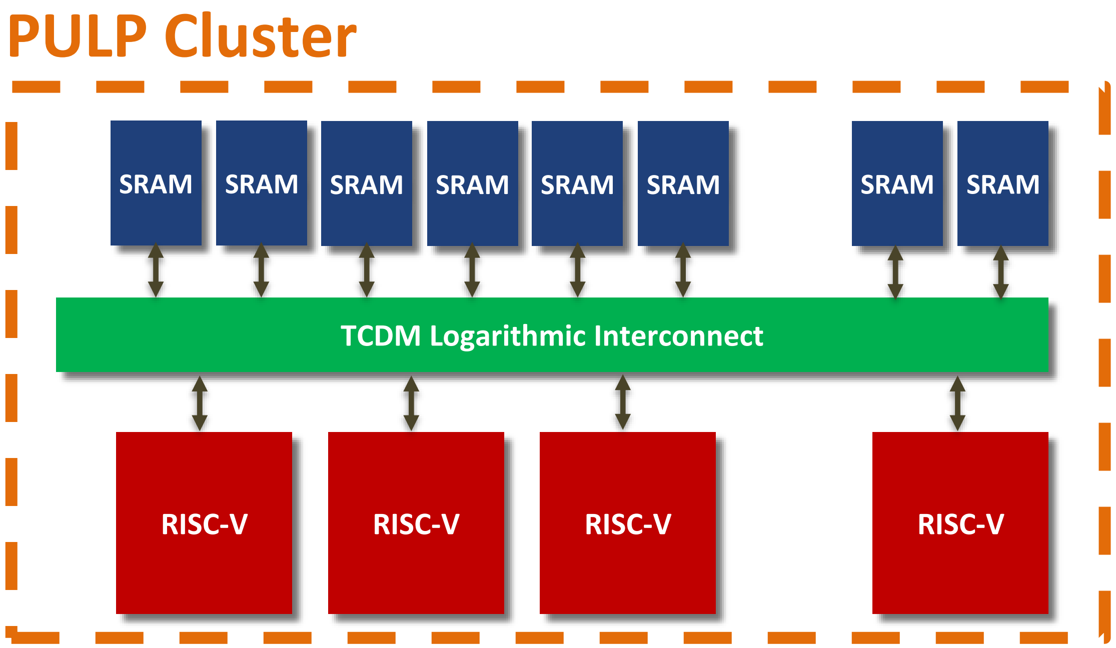


Target a **Shared-Memory** parallel programming model: 4—16 DSP cores sharing directly a **Shared L1 Memory** (***Tightly Coupled Data Mem*** or ***TCDM***)

Organize memory in **Multiple Banks** → concurrent access

PULP: a Parallel Ultra-Low-Power computing platform



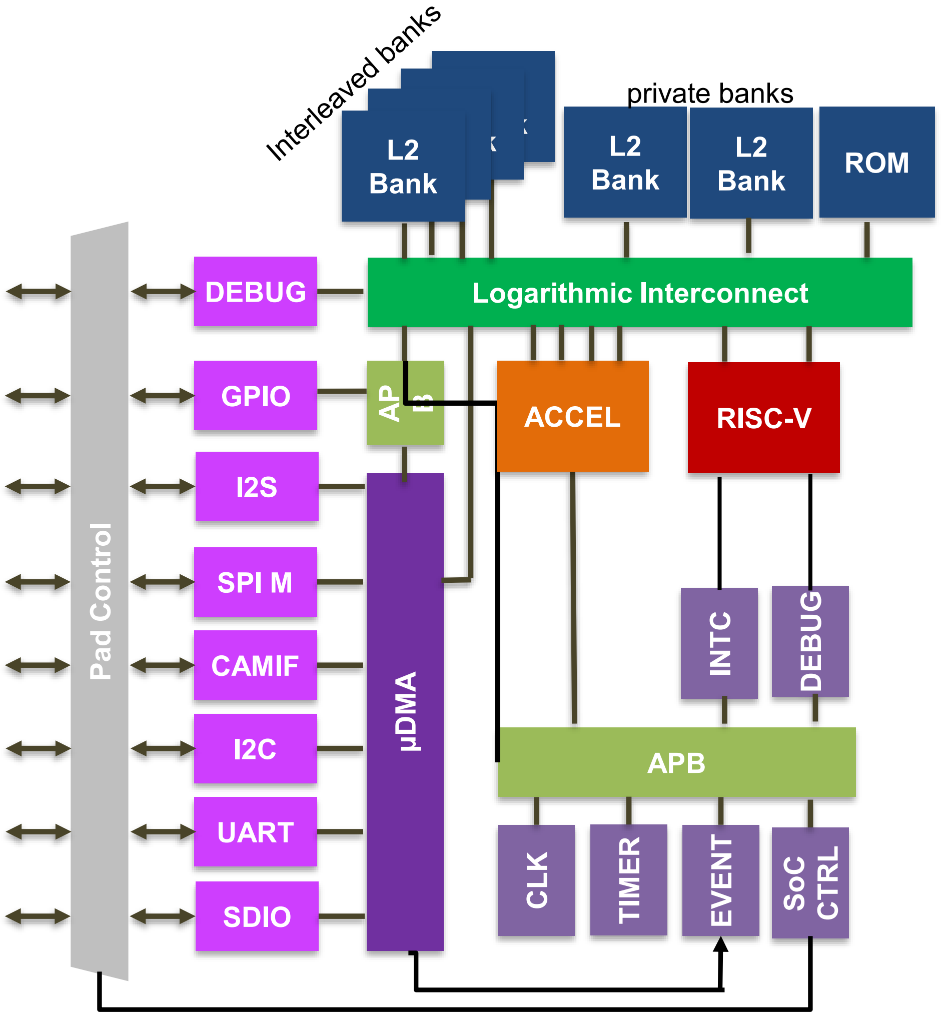


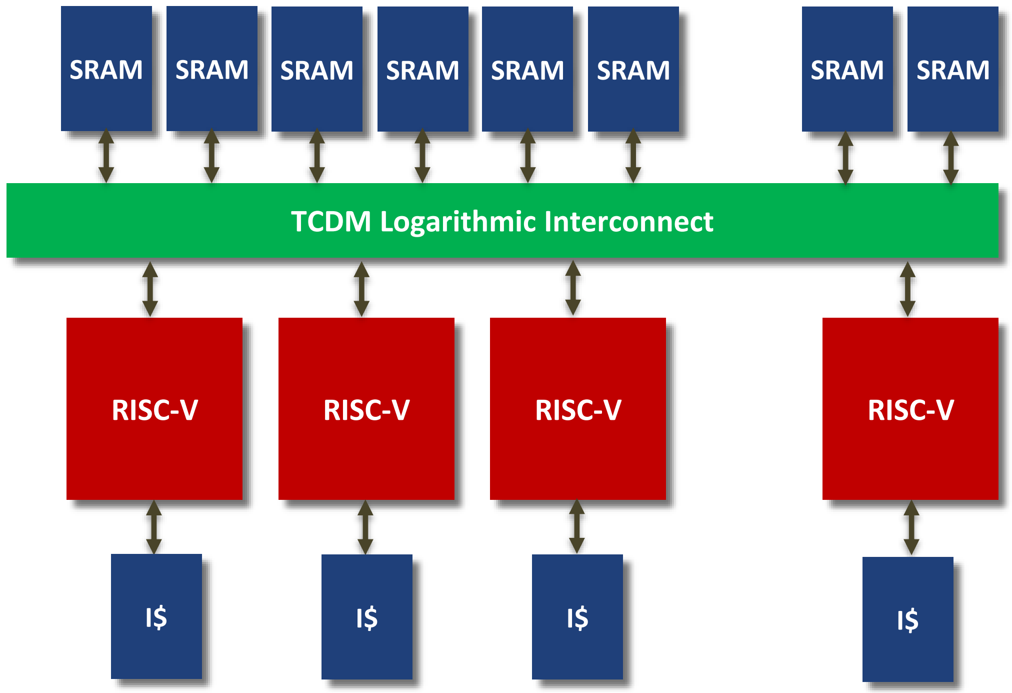
Target a **Shared-Memory** parallel programming model: 4—16 DSP cores sharing directly a **Shared L1 Memory** (***Tightly Coupled Data Mem*** or ***TCDM***)

Organize memory in **Multiple Banks** ⍰ concurrent access

Full connectivity **1-cycle Access Crossbar**

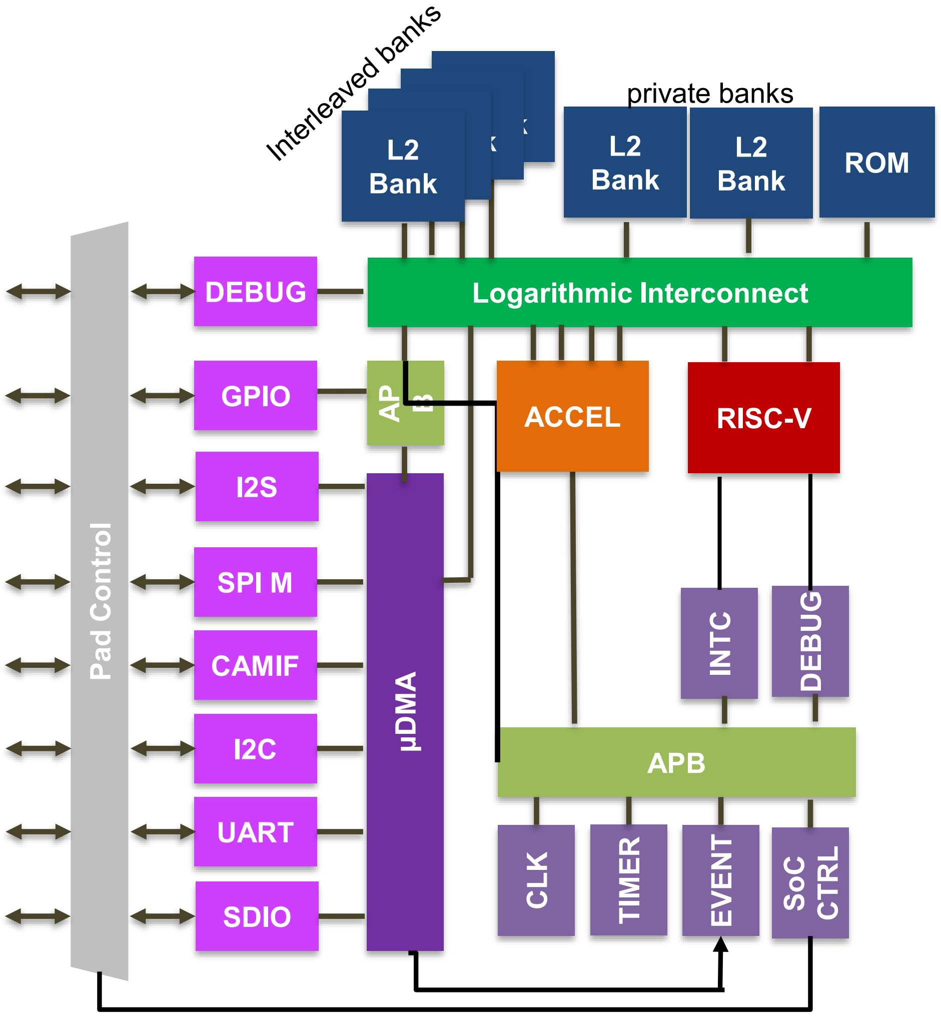
PULP: a Parallel Ultra-Low-Power computing platform

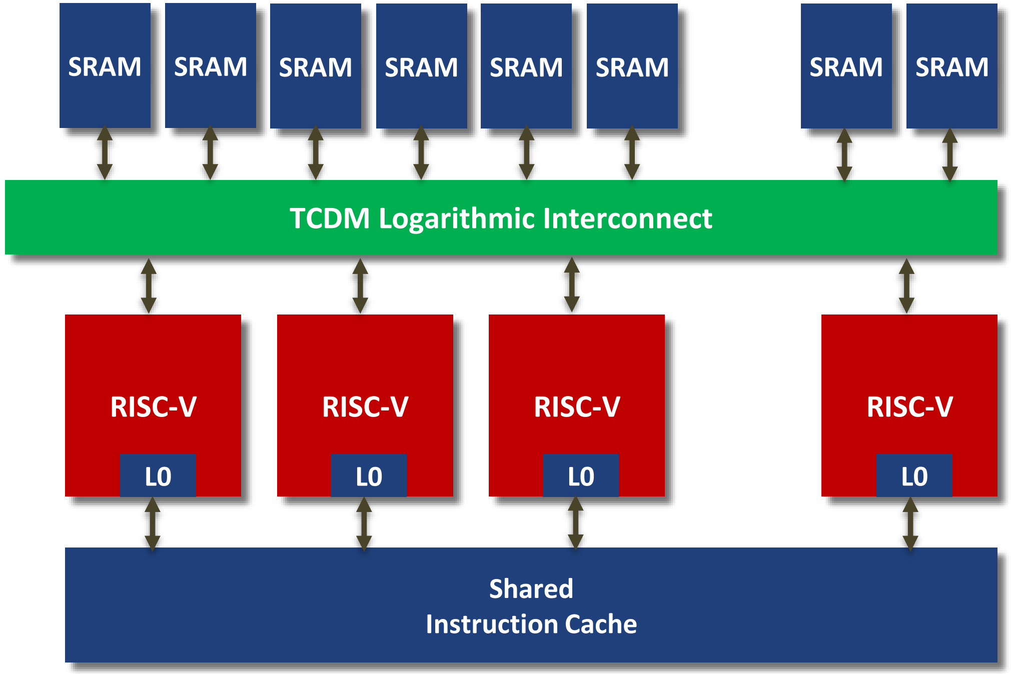




Multiple separate instruction streams (hardware threads): **MIMD** execution scheme...

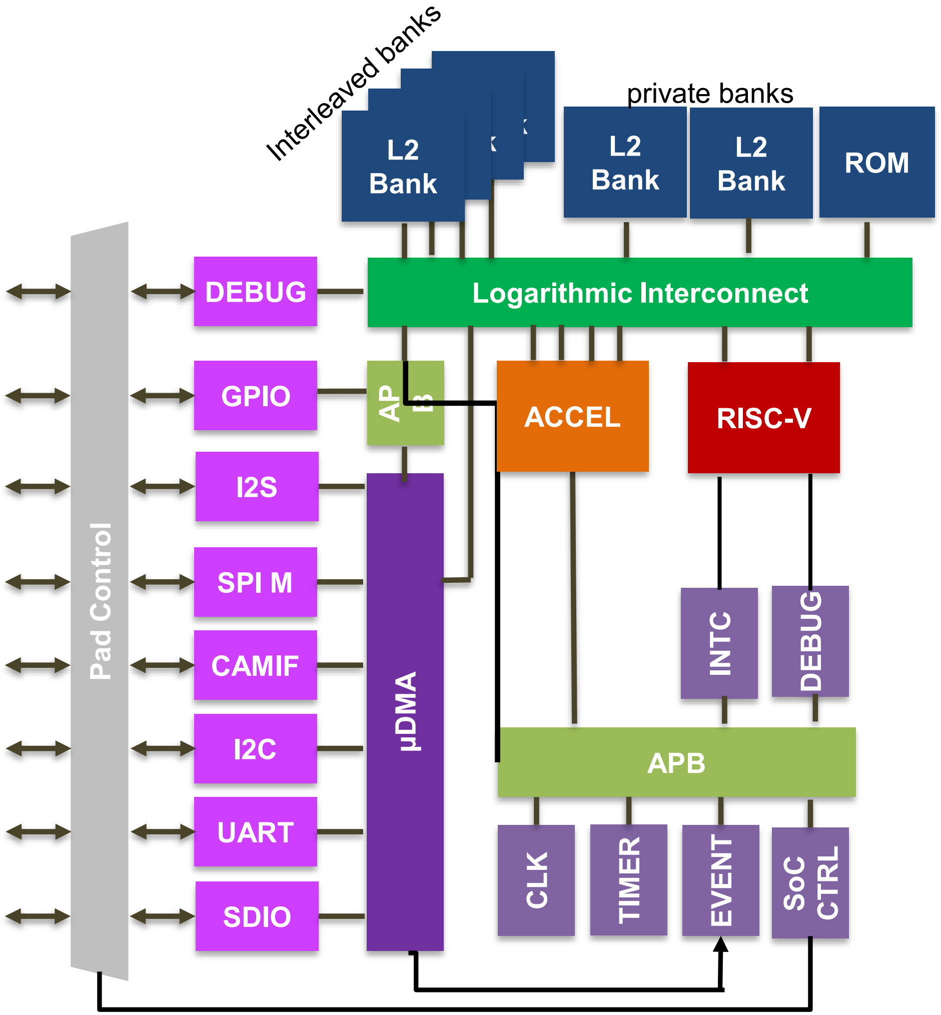
PULP: a Parallel Ultra-Low-Power computing platform

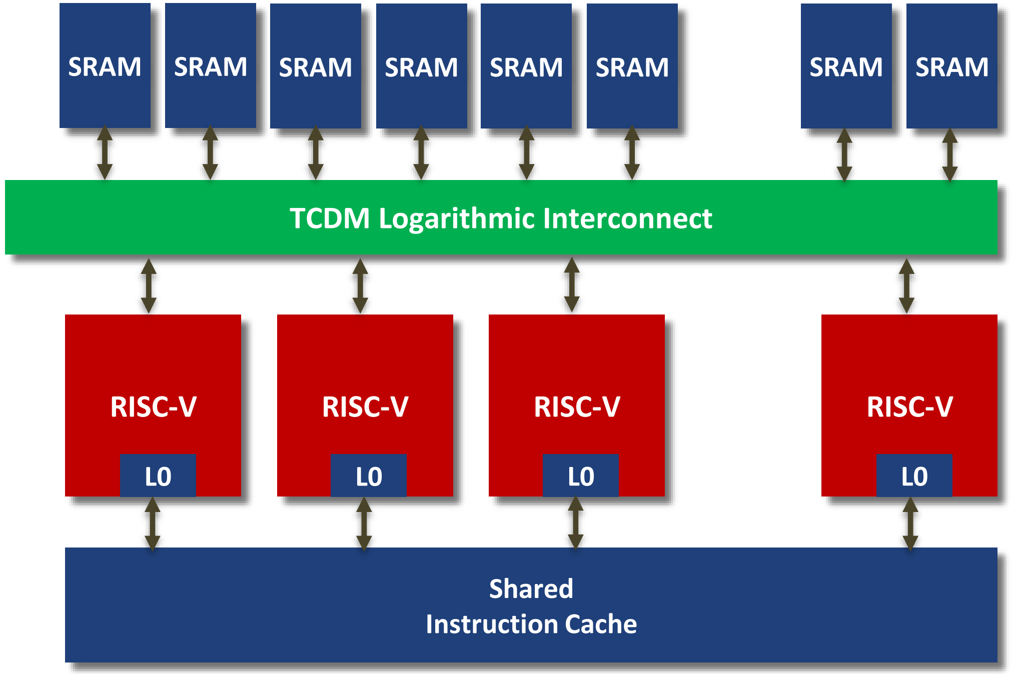




Multiple separate instruction streams (hardware threads):
MIMD execution scheme...
... but optimized for a single parallel program (**SPMD**)

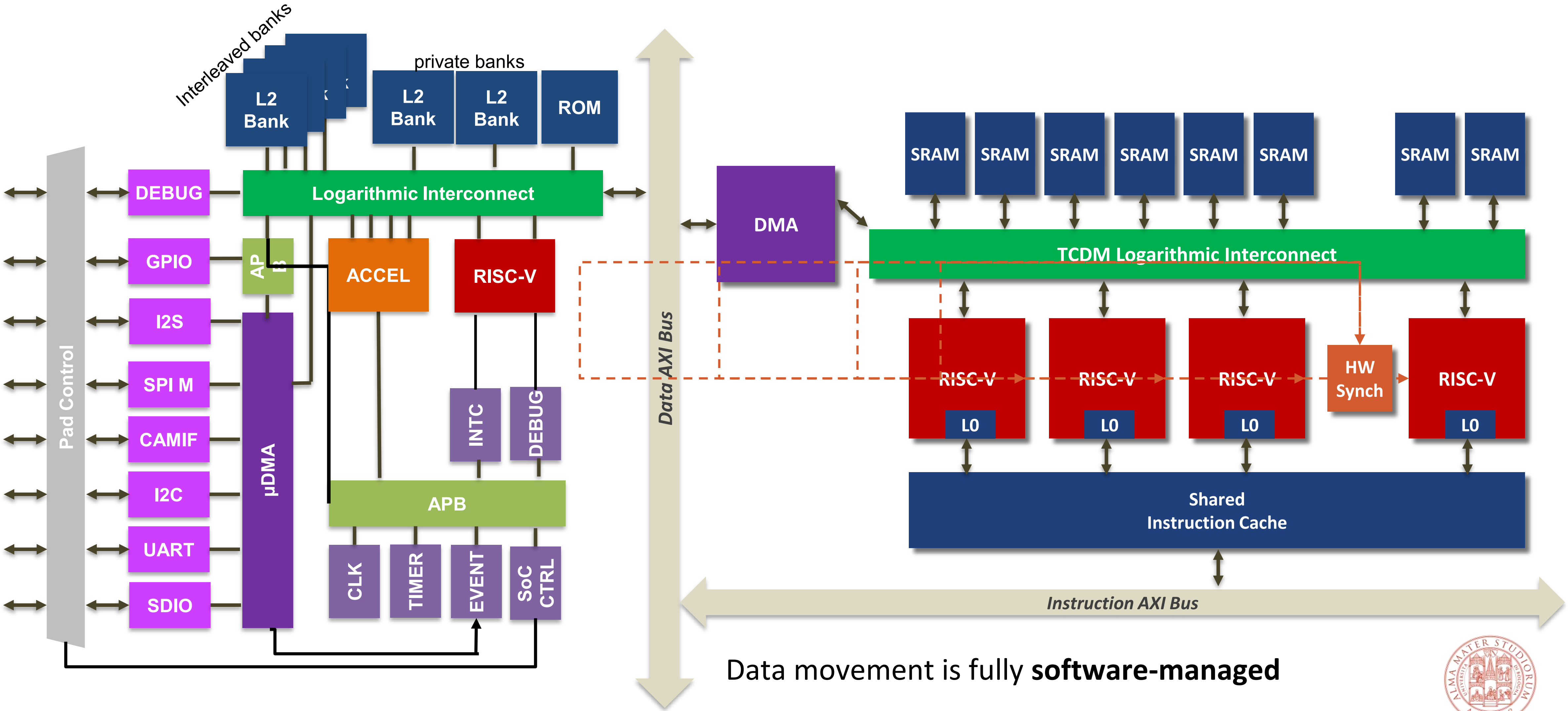
PULP: a Parallel Ultra-Low-Power computing platform





SPMD executes multiple instances of the same program independently, where each program works on a different portion of the data

PULP: a Parallel Ultra-Low-Power computing platform



Data movement is fully **software-managed**

PMSIS: how to manage a device lifecycle

- Configuration and initialization (*device specific*)
  `conf_init()`
  `open_from_conf()`

- Prepare the device for usage:
  `open()`

- Perform required operations (*device specific*)

- Release the resources
  `close()`

PMSIS: Using the cluster as a device

You will see this in main.c

```c
struct pi_device cluster_dev = {0};
struct pi_cluster_conf conf;
struct pi_cluster_task cluster_task = {0};
```

PMSIS data structures

```c
// task allocation
pi_cluster_task(&cluster_task, cluster_entry, NULL);
```

Function pointer

Create a task for the cluster

```c
// init the cluster
pi_cluster_conf_init(&conf);
pi_open_from_conf(&cluster_dev, &conf);
```

Initialize the cluster device

```c
// open the cluster
if (pi_cluster_open(&cluster_dev)) return -1;
```

Open the cluster device

```c
// offload an entry point to the cluster
pi_cluster_send_task_to_cl(&cluster_dev, &cluster_task);
```

Execute code on the cluster

```c
// releasing the cluster
pi_cluster_close(&cluster_dev);
```

Release the cluster device

Executing on the cluster: Fork/join + SPMD

```
static void cluster_entry(void *arg)
{
  printf("Hello from cluster\n");
  // ...

  pi_cl_team_fork(NUM_CORES, cluster_fn, (void *) &args);

  // ...
}
```

Data pointer

Function pointer

Executed on core 0

Fork the execution of the same function on NUM_CORES cores

Executed on core 0

SPMD executes multiple instances of the same program independently, where each program works on a different portion of the data

Synchronization: Barriers and Critical sections

- Barriers (*used for intermediate and final join points*):

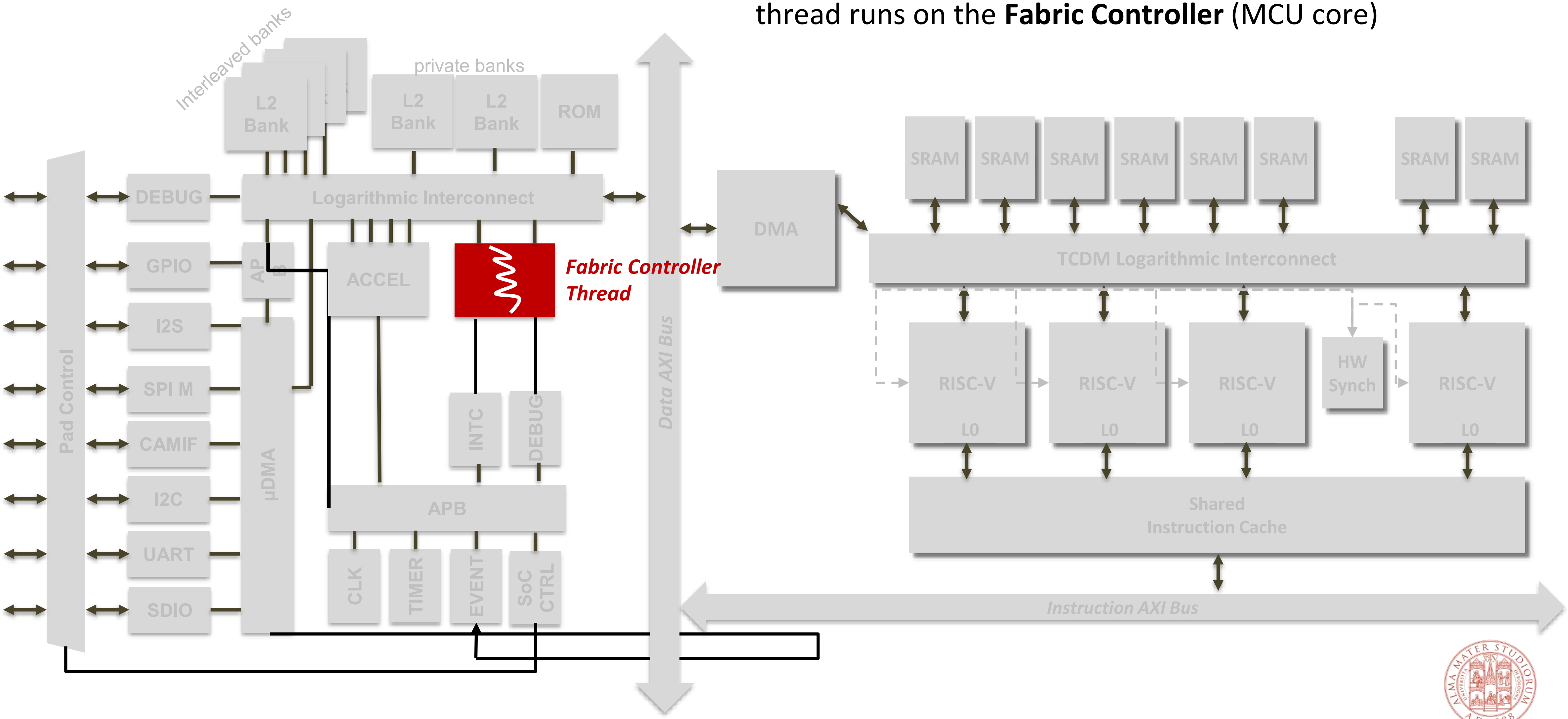
```
pi_cl_team_barrier();
```

- Critical sections (*used to avoid critical races*):

```
pi_cl_team_critical_enter();
// code in the critical section
pi_cl_team_critical_exit();
```

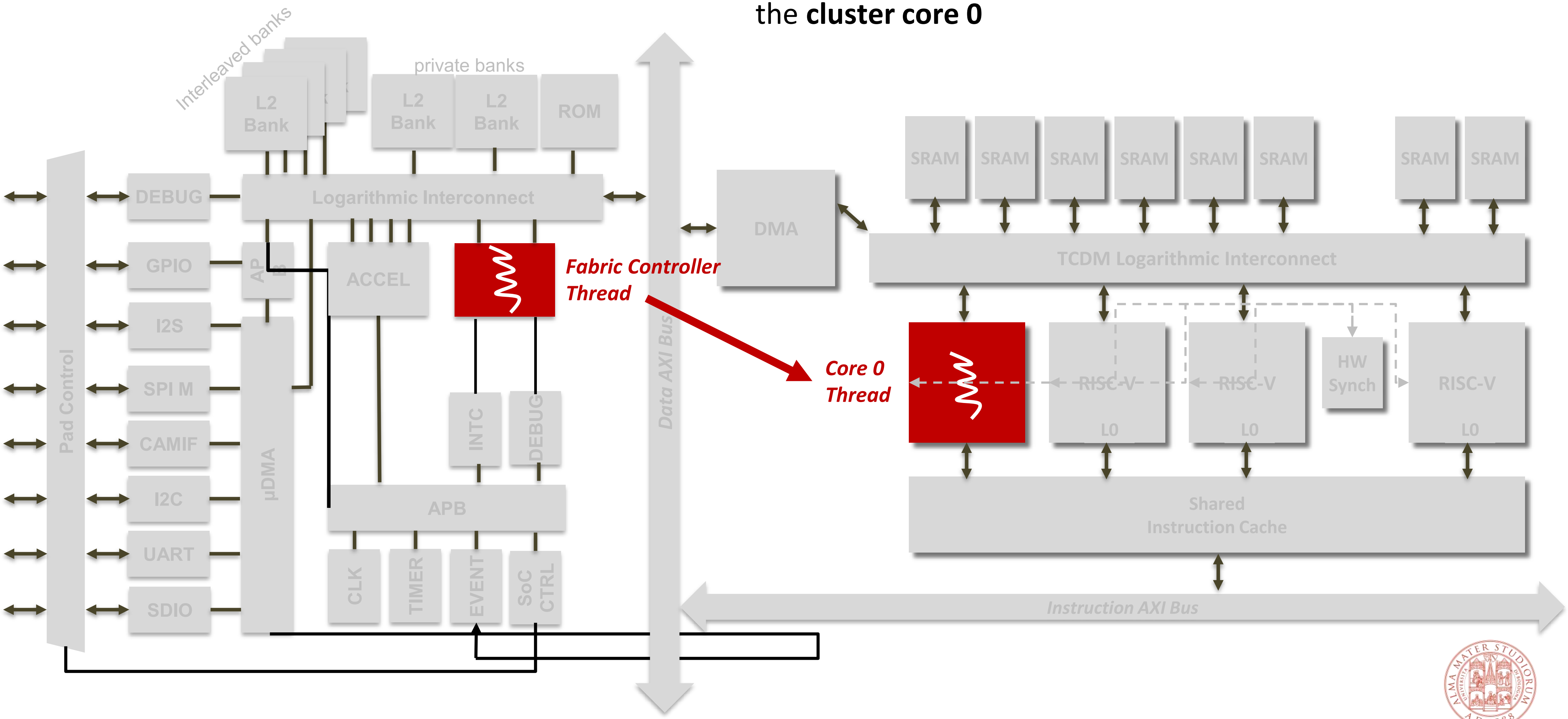
PULP execution model

Cluster is **inactive** and **clock-gated** at boot: a single thread runs on the **Fabric Controller** (MCU core)



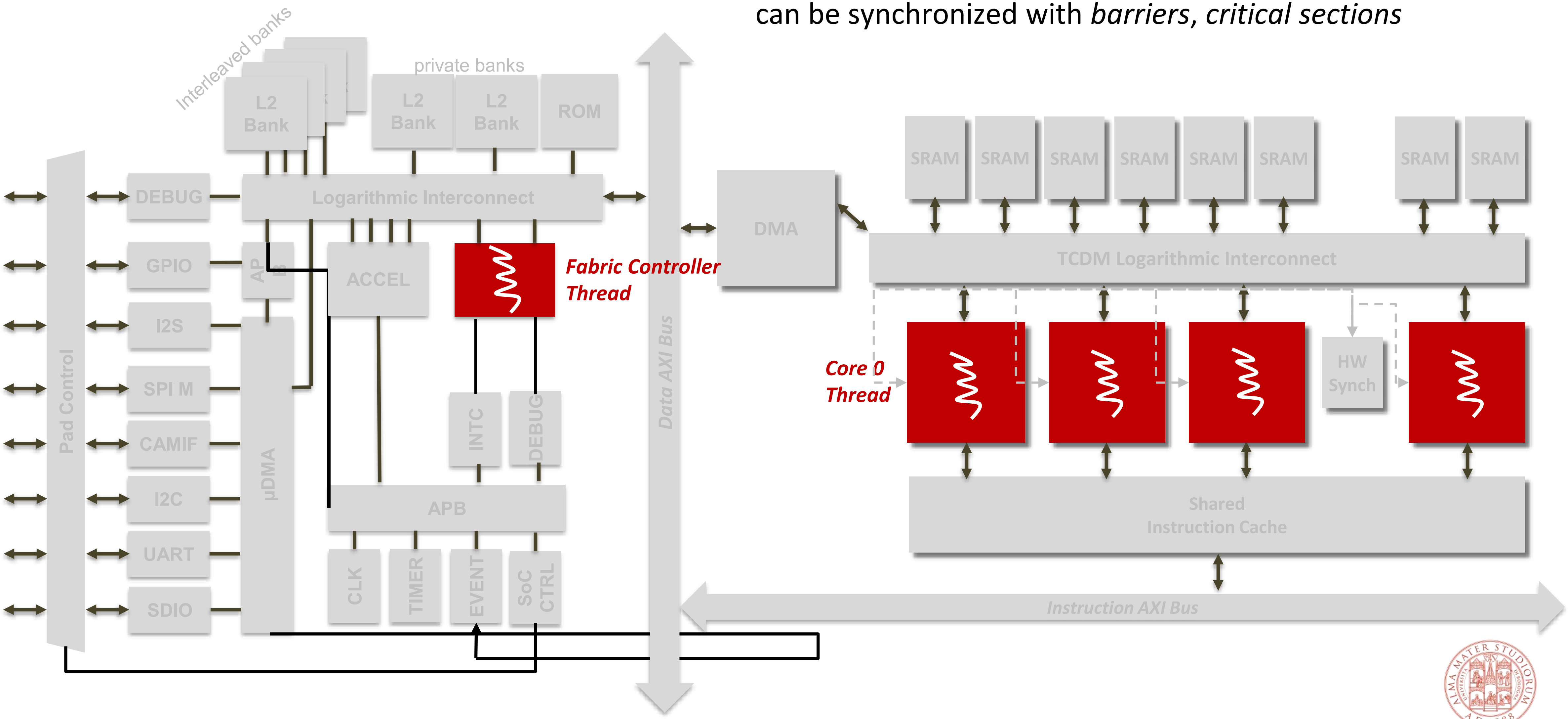
PULP execution model

Activate the cluster (clock it!) and **call** a function on the **cluster core 0**



PULP execution model

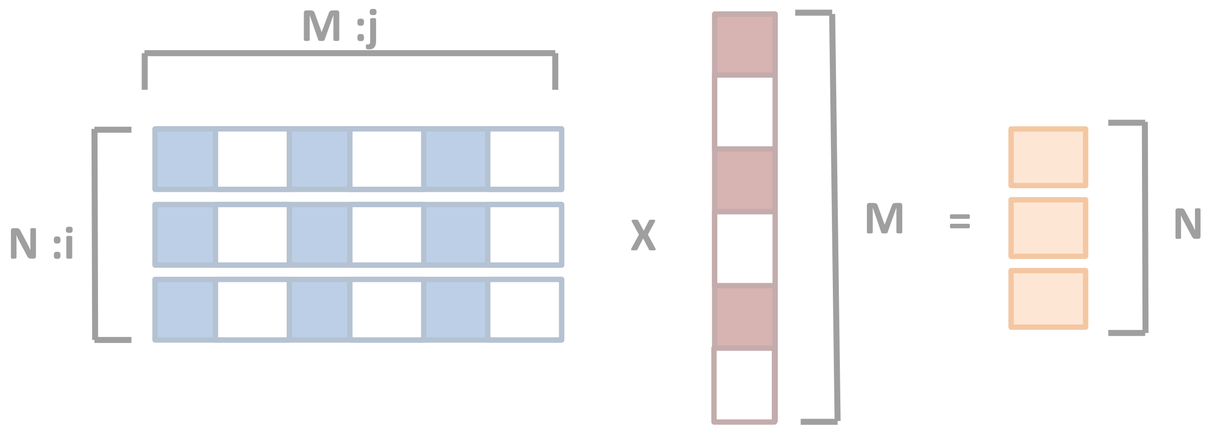
Fork an **execution team** on multiple cluster cores; they can be synchronized with *barriers*, *critical sections*



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

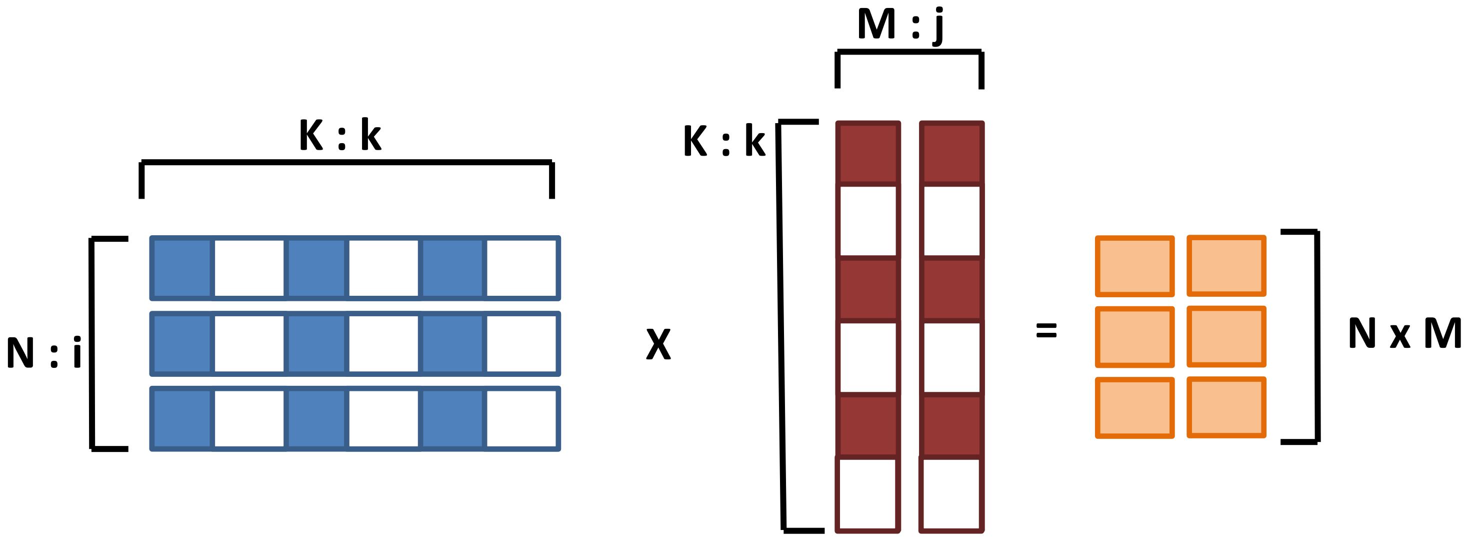
TASK1: MatMul

FROM LAB02: Matrix-vector product



```
// generic matrix-vector multiplication
int gemv(int N, int M, float * mat, float *vec, float * output_vec){

  for (int i=0; i<N; i++){
   for (int j=0; j<M; j++){
    vec_o[i] += mat_i[i*size_M+j] * vec_i[j];
   }
  }

}
```

Today: Matrix Multiplication (MatMul)



```
// generic matrix multiplication
void gemm(int * MatA, int * MatB, int* MatC, int NN, int MM, int KK){
  // task to profile
  for (int i = 0; i < NN; i++) {
   for (int j = 0; j < MM; j++) {
    int acc = 0;
    for (int k = 0; k < KK; k++) {
     acc += MatA[i*KK+k] * MatB[k*MM+j];
    } //k
    MatC[i*MM+j] = acc;
   }//j
  }//i
```

MACs = N*K*M

Matrix Multiplication and Parallelization

Now try to execute the parallelized version of matrix multiplication:

```
$ cd matmul_parallelization/
$ make clean all run CORES=<1 to 8>
```

- Follow the assignment document.

TASK1.1: Matrix Multiplication and Parallelization

```
// generic matrix multiplication
void gemm(int * MatA, int * MatB, int* MatC, int NN, int MM, int KK){
    uint32_t i, core_id, i_chunk, i_start, i_end;

 core_id = pi_core_id();
    i_chunk = (NN + NUM_CORES-1) / NUM_CORES;
    i_start = core_id * i_chunk;
    i_end   = i_start + i_chunk < NN ? i_start + i_chunk : NN;

    // task to profile
    for (i = i_start; i < i_end; i ++) {
      for (int j = 0; j < MM; j++) {
        int acc = 0;
        for (int k = 0; k < KK; k++) {
          acc += MatA[i*KK+k] * MatB[k*MM+j];
        } //k
        MatC[i*MM+j] = acc;
      }//j
    }//I
    pi_cl_team_barrier();
}
```

Divide the workload in adjacent blocks (*chunks*) and compute their bounds [i_start, i_end]

We talked before about SPMD:
executes multiple instances of the same program independently, where each program works on a different portion of the data

TASK1.1: Matrix Multiplication and Parallelization

```
// generic matrix multiplication
void gemm(int * MatA, int * MatB, int* MatC, int NN, int MM, int KK){
    uint32_t i, core_id, i_chunk, i_start, i_end;

 core_id = pi_core_id();
    i_chunk = (NN + NUM_CORES-1) / NUM_CORES;
    i_start = core_id * i_chunk;
    i_end   = i_start + i_chunk < NN ? i_start + i_chunk : NN;

    // task to profile
    for (i = i_start; i < i_end; i ++) {
      for (int j = 0; j < MM; j++) {
        int acc = 0;
        for (int k = 0; k < KK; k++) {
          acc += MatA[i*KK+k] * MatB[k*MM+j];
        } //k
        MatC[i*MM+j] = acc;
      }//j
    }//I
    pi_cl_team_barrier();
}
```

Divide the workload in adjacent blocks (*chunks*) and compute their bounds [i_start, i_end]

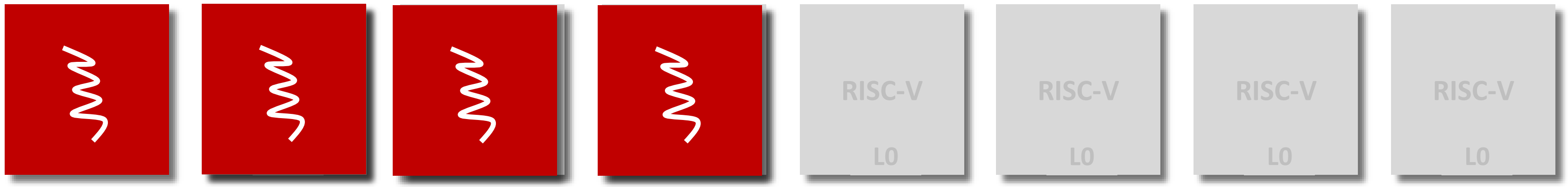
We talked before about SPMD:
executes multiple instances of the same program independently, where each program works on a different portion of the data

core_id is used to divide portion of the data among all cores

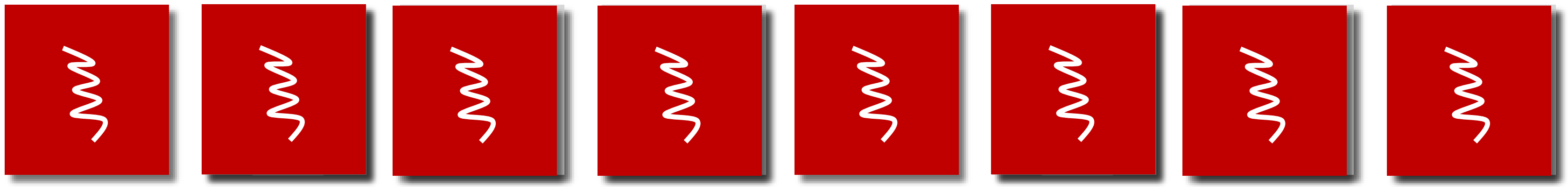
TASK1.2: different input sizes

- We parallelize on the number of rows N.
- We have 8 cores

If the number f rows is 4: → only 4 cores busy



If the number f rows is 8 → 8 cores busy



Task 1.3: Matrix Multiplication and Parallelization w/ Manual Unrolling

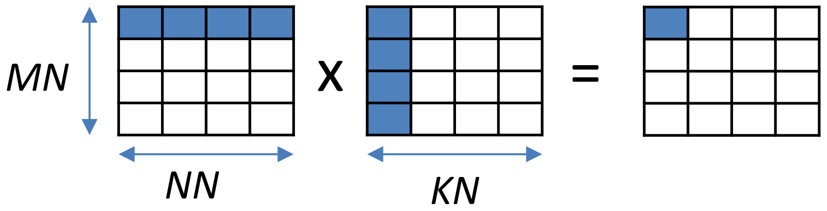
Manual (or static) **loop unrolling** helps reducing the total number of instructions!.

```
void gemm_unroll_1x4(int * MatA, int * MatB, int* MatC, int
NN, int MM, int KK){

  uint32_t core_id, i_chunk, i_start, i_end;
  uint32_t i = 0;

  core_id = pi_core_id();
  i_chunk = (NN + NUM_CORES-1) / NUM_CORES;
  i_start = core_id * i_chunk;
  i_end   = i_start + i_chunk < NN ? i_start + i_chunk :
NN;

                  < unrolled loop here >
}
```

```
…
for (i = i_start; i < i_end; i ++) {
 for (int j = 0; j < MM; j=j+4) {
   int acc0 = 0;
   int acc1 = 0;
   int acc2 = 0;
   int acc3 = 0;
   for (int k = 0; k < KK; k++) {
      int shared_op = MatA[i*KK+k];
      int idx = k*MM+j;
      acc0   += shared_op * MatB[idx];
      acc1   += shared_op * MatB[idx+1];
      acc2   += shared_op * MatB[idx+2];
      acc3   += shared_op * MatB[idx+3];
   } //k
   MatC[i*MM+j] = acc0;
   MatC[i*MM+j+1] = acc1;
   MatC[i*MM+j+2] = acc2;
   MatC[i*MM+j+3] = acc3;
 }//j
}//i
```

Since you unroll **4 operations**, you should cycle **4 by 4**.

Matrix Multiplication: Baseline



Internal loop

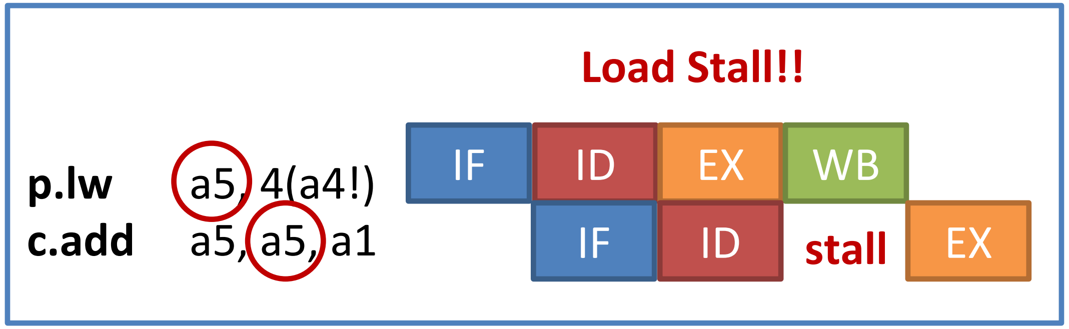
4096 (16x16x16) iterations

4 instructions ≈ **16834** ops

```
for (int i = 0; i < MN; i++) {
   for (int j = 0; j < NN; j++) {
      int acc = 0;
      for (int k = 0; k < KN; k++) {
          acc += in1[i*KN+k] * in2[k*NN+j];
      } //k
      out3[i*NN+j] = acc;
   }//j
}//i
```

```
lp.setup      x1,a6,1c0089e4

lbu           t4,0(t1)    // load in2
p.lbu         t5,1(t3!)   // load in1
add           t1,t1,a4    // in2 incr
p.mac         a7,t5,t4    // out += in2*in1
```

2 LOAD + 1 ADD + 1 MAC


Load Stall!!

p.lw a5, 4(a4!)

c.add a5, a5, a1

$$MN = NN = KN = \mathbf{16}$$

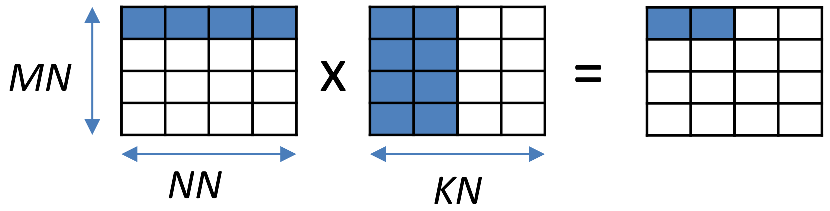
$$\boldsymbol{MAC} = 16^3 = 4096$$

Number of Instructions:	**18867**
Clock Cycles:	**18911**
CPI:	**1.00**
MAC/cyc:	**0.21**

Matrix Multiplication: Unrolling I

Loop Unrolling to enable data reuse and reduce number of load instructions!

```
for (int i = 0; i < MN; i++) {
   for (int j = 0; j < NN; j=j+2) {
      int acc0 = 0;
      int acc1 = 0;
      for (int k = 0; k < KN; k++) {
         char shared_op = in1[i*KN+k];
         acc0   += shared_op * in2[k*NN+j];
         acc1   += shared_op * in2[k*NN+j+1];
      } //k
      out3[i*NN+j] = acc0;
      out3[i*NN+j+1] = acc1;
   } // loop j
 } // loopi
```



Internal loop

2048 (16x**8**x16) iterations

7 instructions ≈ **14336** ops

```
p.lbu        t3,1(t6!)
lbu          t2,0(t1)
lbu          t0,0(a7)
add          t1,t1,a4
p.mac        t4,t3,t2
add          a7,a7,a4
p.mac        t5,t3,t0
```

3 LOAD + 2 ADD + 2 MAC

Number of Instructions: **16282**

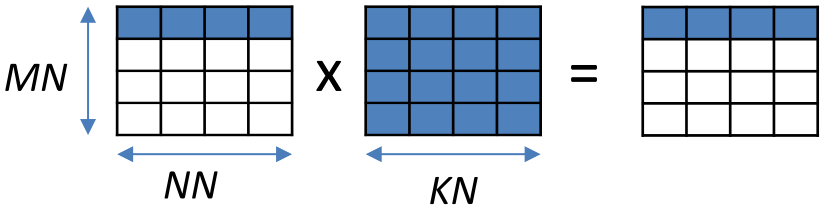
Clock Cycles: **16326**

CPI: **1.00**

MAC/cyc: **0.25**

1.15x!

Matrix Multiplication: Unrolling II



```
for (int i = 0; i < MN; i++) {
   for (int j = 0; j < NN; j=j+4) {
      int acc0 = 0;
      int acc1 = 0;
      int acc2 = 0;
      int acc3 = 0;
      for (int k = 0; k < KN; k++) {
         char shared_op = in1[i*KN+k];
         acc0   += shared_op * in2[k*NN+j];
         acc1   += shared_op * in2[k*NN+j+1];
         acc2   += shared_op * in2[k*NN+j+2];
         acc3   += shared_op * in2[k*NN+j+3];
      } //k
      out3[i*NN+j]   = acc0;
      out3[i*NN+j+1] = acc1;
      out3[i*NN+j+2] = acc2;
      out3[i*NN+j+3] = acc3;
   }//j
}//i
```

Internal loop

1024 (16x4x16) iterations

10 instructions ≈ **10240** ops

```
p.lbu    a7,1(t0!)
lbu      s0,0(a6)
lbu      a0,1(a6)
lbu      a2,2(a6)
lbu      t2,3(a6)
p.mac    t3,a7,s0
add      a6,a6,a4
p.mac    t4,a7,a0
p.mac    t5,a7,a2
p.mac    t6,a7,t2
```

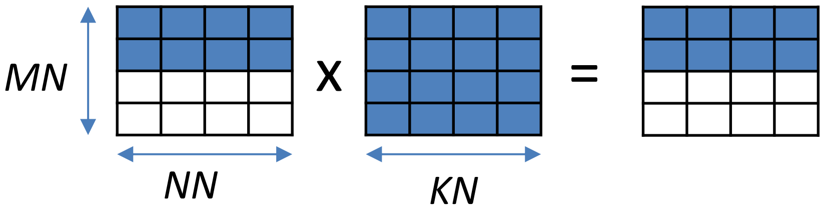
5 LOAD + 1 ADD + 4MAC

Number of Instructions:	**11579**
Clock Cycles:	**11719**
CPI:	**1.01**
MAC/cyc:	**0.35**

1.4x!

Matrix Multiplication: Unrolling III

```
for (int i = 0; i < MN; i=i+2) {
    for (int j = 0; j < NN; j=j+4) {
      acc0 = acc1 = acc2 = acc3 = 0;
      acc4 = acc5 = acc6 = acc7 = 0;

      for (int k = 0; k < KN; k++) {
         char shared_op = in1[i*KN+k];
         char a = in2[k*NN+j];
         char b = in2[k*NN+j+1];
         char c = in2[k*NN+j+2];
         char d = in2[k*NN+j+3];

         acc0   += shared_op * a;
         acc1   += shared_op * b;
         acc2   += shared_op * c;
         acc3   += shared_op * d;

         shared_op = in1[(i+1)*KN+k];
         acc4   += shared_op * a;
         acc5   += shared_op * b;
         acc6   += shared_op * c;
         acc7   += shared_op * d;
      } //k
      out3[i*NN+j]    = acc0;        out3[i*NN+j+1] = acc1;
      out3[i*NN+j+2] = acc2;         out3[i*NN+j+3] = acc3;
      out3[(i+1)*NN+j]    = acc4;    out3[(i+1)*NN+j+1] = acc5;
      out3[(i+1)*NN+j+2] = acc6;     out3[(i+1)*NN+j+3] = acc7;
    }//j
  }//I
```



Internal loop

512 (8x4x16) iterations

15 instructions ≈ **7680** ops

```
p.lbu    a2,1(s2!)       p.mac    t4,a2,a7
p.lbu    a3,1(s3!)       p.mac    t5,a2,a6
lbu      t1,0(a5)        p.mac    t6,a2,a0
lbu      a7,1(a5)        p.mac    t0,t1,a3
lbu      a6,2(a5)        p.mac    t2,a7,a3
lbu      a0,3(a5)        p.mac    s0,a6,a3
p.mac    t3,a2,t1        p.mac    s1,a0,a3
add      a5,a5,a4
```

6 LOAD + 1 ADD + 8 MAC

Number of Instructions: **8768**

Clock Cycles: **8812**

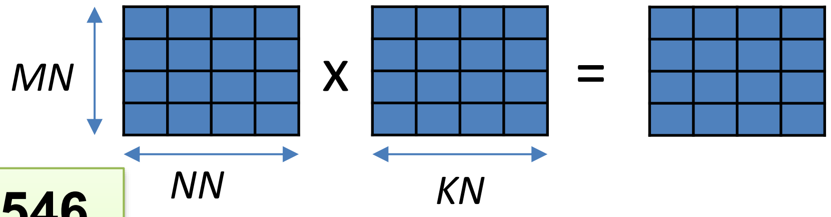
CPI: **1.01**

MAC/cyc:

0.464

1.33x!

Matrix Multiplication: Unrolling IV



```
for (int i = 0; i < MN; i=i+4) {
      for (int j = 0; j < NN; j=j+4) {
        ...
        for (int k = 0; k < KN; k++) {
           char shared_op = in1[i*KN+k];
           char a = in2[k*NN+j];
           char b = in2[k*NN+j+1];
           char c = in2[k*NN+j+2];
           char d = in2[k*NN+j+3];

           acc0   += shared_op * a;
           acc1   += shared_op * b;
           acc2   += shared_op * c;
           acc3   += shared_op * d;

           shared_op = in1[(i+1)*KN+k];
           acc4   += shared_op * a;
           acc5   += shared_op * b;
           acc6   += shared_op * c;
           acc7   += shared_op * d;

           shared_op = in1[(i+2)*KN+k];
           acc8   += shared_op * a;
           acc9   += shared_op * b;
           acc10  += shared_op * c;
           acc11  += shared_op * d;

           shared_op = in1[(i+3)*KN+k];
           acc12  += shared_op * a;
           acc13  += shared_op * b;
           acc14  += shared_op * c;
           acc15  += shared_op * d;
      }
```

Number of Instructions:	**10546**
Clock Cycles:	**11342**
CPI:	**1.07**
MAC/cyc:	**0.361**

```
…
p.lbu    a7,1(s9!)
p.lbu    a6,1(s10!
p.lbu    a4,1(s11!
sw       t4,8(sp)
lw       t4,40(sp)
p.mac    s1,a2,a7
add      a5,a5,t4
lw       t4,20(sp)
p.mac    t2,a0,a7
p.mac    s0,a1,a7
p.mac    s2,a3,a7
lw       a7,16(sp)
p.mac    t4,a0,a4
p.mac    a7,a3,a6
sw       t4,20(sp)
p.mac    t6,t1,a2
…
```

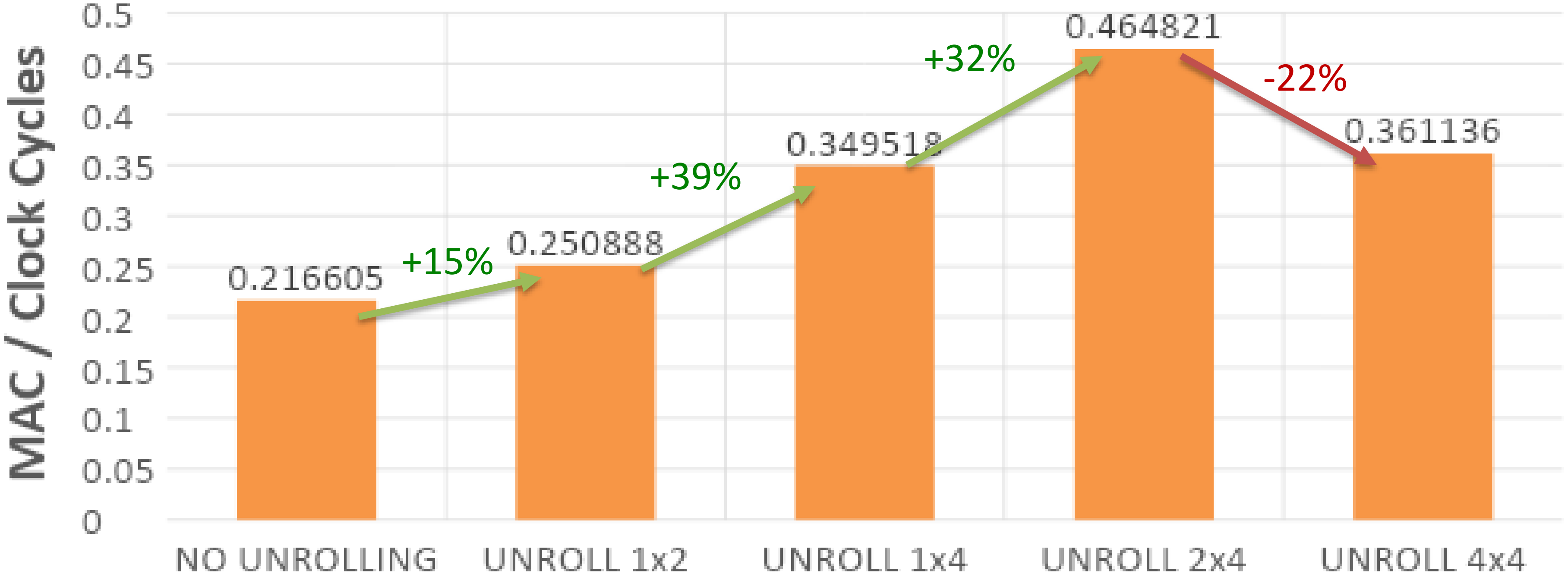
The register file is not infinite! ☹

For wide loop body, the compiler may cannot find **enough registers to map all the required variables**.

If this happens, variables start getting stored on the *stack* (***spilling***)

- Performance drop!

Recap: Loop Unrolling of MatMult

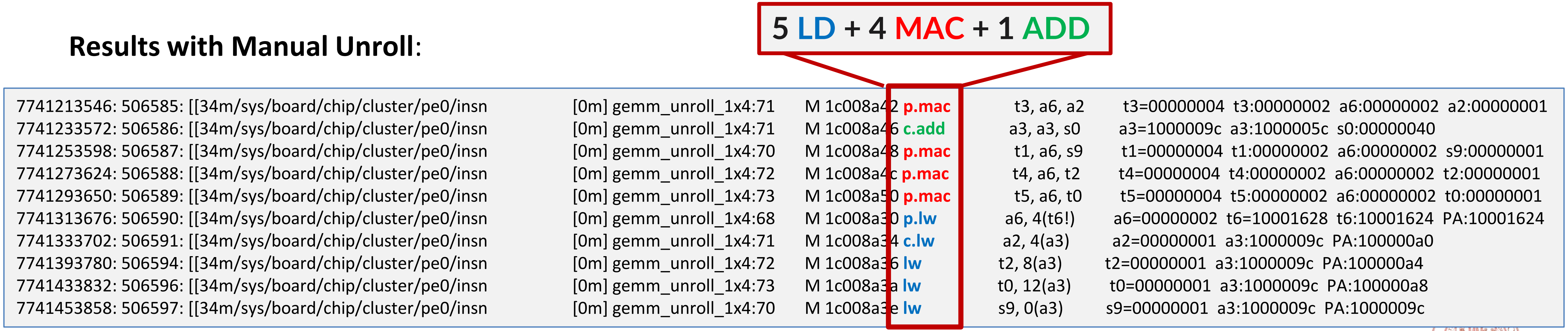


Note: assumption of *MN*/*NN* multiple of 2/4. *What if* this was not the case?

Disassembly code for unrolled matmul: no stalls

1. Parallel MatMul w/ manual unrolling.
   - Replace *gemm* with *gemm_unroll*
   - Plot #instr and #clk by varying CORES
   - You can see it in the disassembly

Results with Manual Unroll:



5 LD + 4 MAC + 1 ADD

```
7741213546: 506585: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:71   M 1c008a42 p.mac    t3, a6, a2     t3=00000004  t3:00000002  a6:00000002  a2:00000001
7741233572: 506586: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:71   M 1c008a46 c.add    a3, a3, s0     a3=1000009c  a3:1000005c  s0:00000040
7741253598: 506587: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:70   M 1c008a48 p.mac    t1, a6, s9     t1=00000004  t1:00000002  a6:00000002  s9:00000001
7741273624: 506588: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:72   M 1c008a4c p.mac    t4, a6, t2     t4=00000004  t4:00000002  a6:00000002  t2:00000001
7741293650: 506589: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:73   M 1c008a50 p.mac    t5, a6, t0     t5=00000004  t5:00000002  a6:00000002  t0:00000001
7741313676: 506590: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:68   M 1c008a30 p.lw     a6, 4(t6!)     a6=00000002  t6=10001628  t6:10001624  PA:10001624
7741333702: 506591: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:71   M 1c008a34 c.lw     a2, 4(a3)      a2=00000001  a3:1000009c  PA:100000a0
7741393780: 506594: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:72   M 1c008a36 lw       t2, 8(a3)      t2=00000001  a3:1000009c  PA:100000a4
7741433832: 506596: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:73   M 1c008a3a lw       t0, 12(a3)     t0=00000001  a3:1000009c  PA:100000a8
7741453858: 506597: [[34m/sys/board/chip/cluster/pe0/insn     [0m] gemm_unroll_1x4:70   M 1c008a3e lw       s9, 0(a3)      s9=00000001  a3:1000009c  PA:1000009c
```

Write parallel code

Try on your own!

1. Parallel MatMul w/ manual unrolling.
   - Create a new folder (*cp –r matmul_parallelization/ matmul_unroll/)*
   - Replace *gemm* with *gemm_unroll_1x4*
   - Adjust gemm_unroll_1x4 to combine loop unrolling with parallelization
   - Plot #instr and #clk by varying CORES
   - Analyze your measurements against disassembly (do the measured number of #instructions and #clk make sense?)

Results with Manual Unroll:

Number of Instructions: 33097
Clock Cycles: 33956 | **1 core** execution

Number of Instructions: 4190
Clock Cycles: 4548 | **8 cores** execution

S.U. = 7.90

Use traces to verify the amount of INSTR and CYC:

```
$ make clean all run runner_args='--trace=cluster/pe0/insn:trace.txt'
```

If you perform a ctrl+f in the **trace file**, searching for your **gemm** function name, you will see a corresponding number of instructions!

Reference on Makefile rules

The Makefile is a “recipe” used to call the compiler/linker and in this case also to run the program. You can also chain several rules (e.g., **make clean all run**) and pass options (e.g., **runner_args="--vcd"**)

Remove previous build

```
make clean
```

Build program (calling compiler + linker)

```
make all
```

Run the program

```
make run
```

Run options: you can change them by adding **runner_args="*OPTIONS*"**

```
make run runner_args="--vcd --event=.*"                # visual trace in GTKwave
make run runner_args="--trace=.*insn.*" > trace.log    # written trace of instructions
```

To open GTKwave, run the line that is visible in the log (**gtkwave some_long_string.gtkwave**) in the terminal

Disassembling:

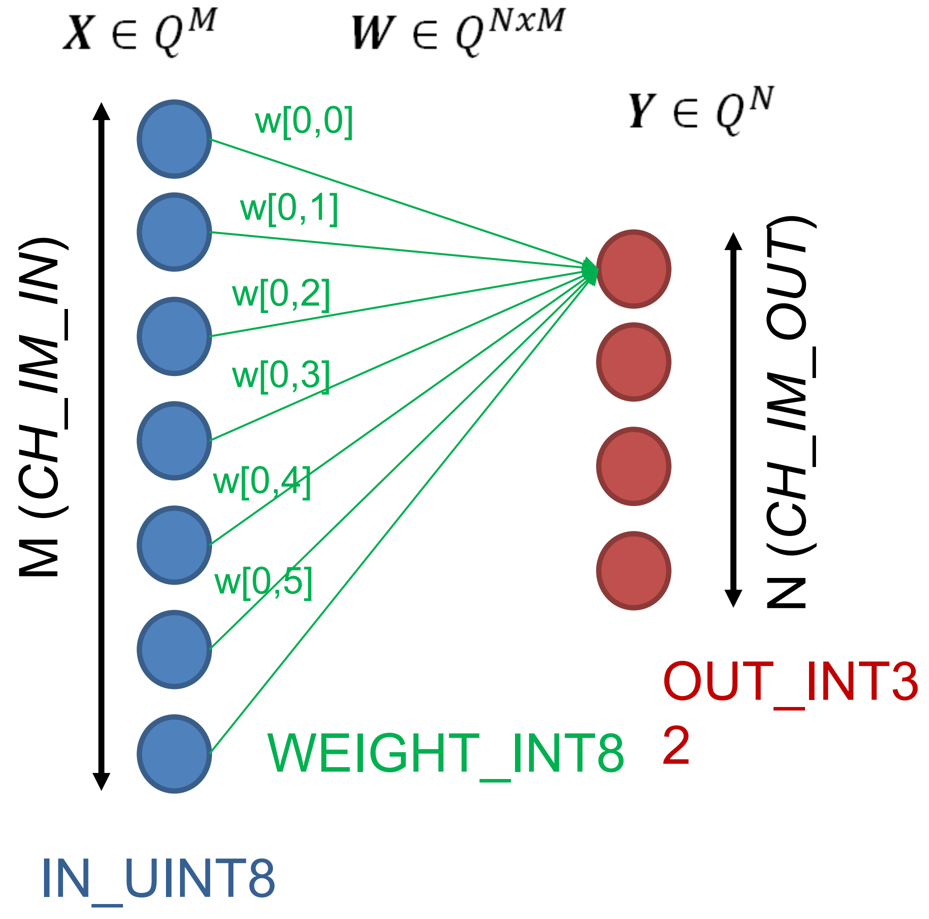
```
 make dis > test.S                                                    # disassemble without inlined source code
/pulp/pkg/pulp_riscv_gcc/bin/riscv32-unkown-elf-objdump -D -S > test.S   # disassemble with inlined source code
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

TASK2: FC

DNN Fully-Connected Layer on PULP

```
$ cd fully_connected/
$ make clean all run CORES=<1 to 8>
```



int32 int8

$$\boxed{Y = W \cdot X} \rightarrow y[i] = \sum_{k=0}^{M} w[i,k] \cdot x[k]$$

matrix-vector multiplication

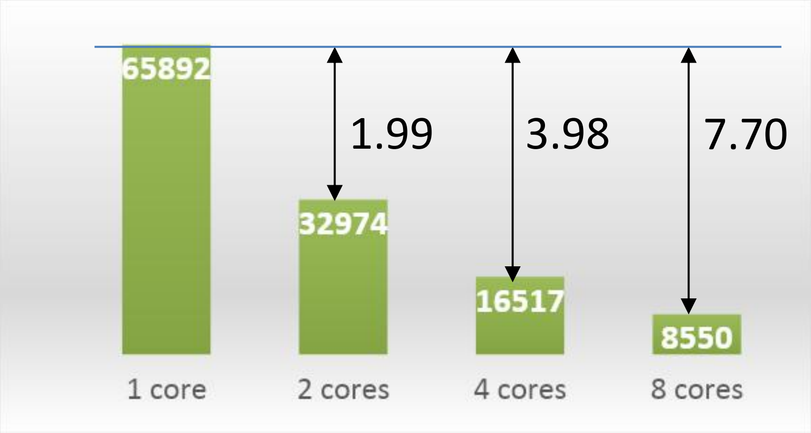
MAC operation!

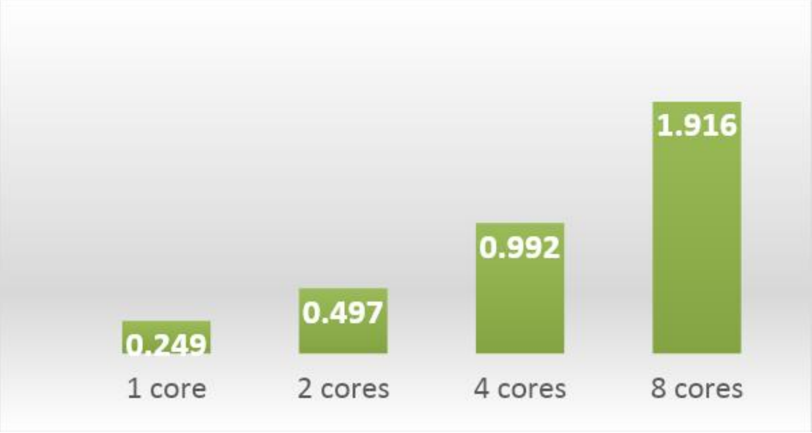
- How many MACs per layer?
- How many cycles on a single core?
- How many cycles on a multi core platform?
  - Compute the speedup
- How many instructions? Check against the assembly code!

```
$ make dis
```

Benchmarking the Fully-Connected Layer

Number of MAC =
M x N =
CH_IM_IN x
CH_IM_OUT =
16384





# cores	#clk	#clk/ MAC	#instr	# instr/ MAC
1				
2				
4				
8				

```
1c008bd8 <pulp_nn_linear_u8_i32_i8>:
……
1c008c2e:     0067c0fb     lp.setup     x1,a5,1c008c3a <pulp_nn_linear_u8_i32_i8+0x62>
1c008c32:     00134e8b     p.lbu        t4,1(t1!)
1c008c36:     00188e0b     p.lb         t3,1(a7!)
1c008c3a:     43ce8833     p.mac        a6,t4,t3
…..
```

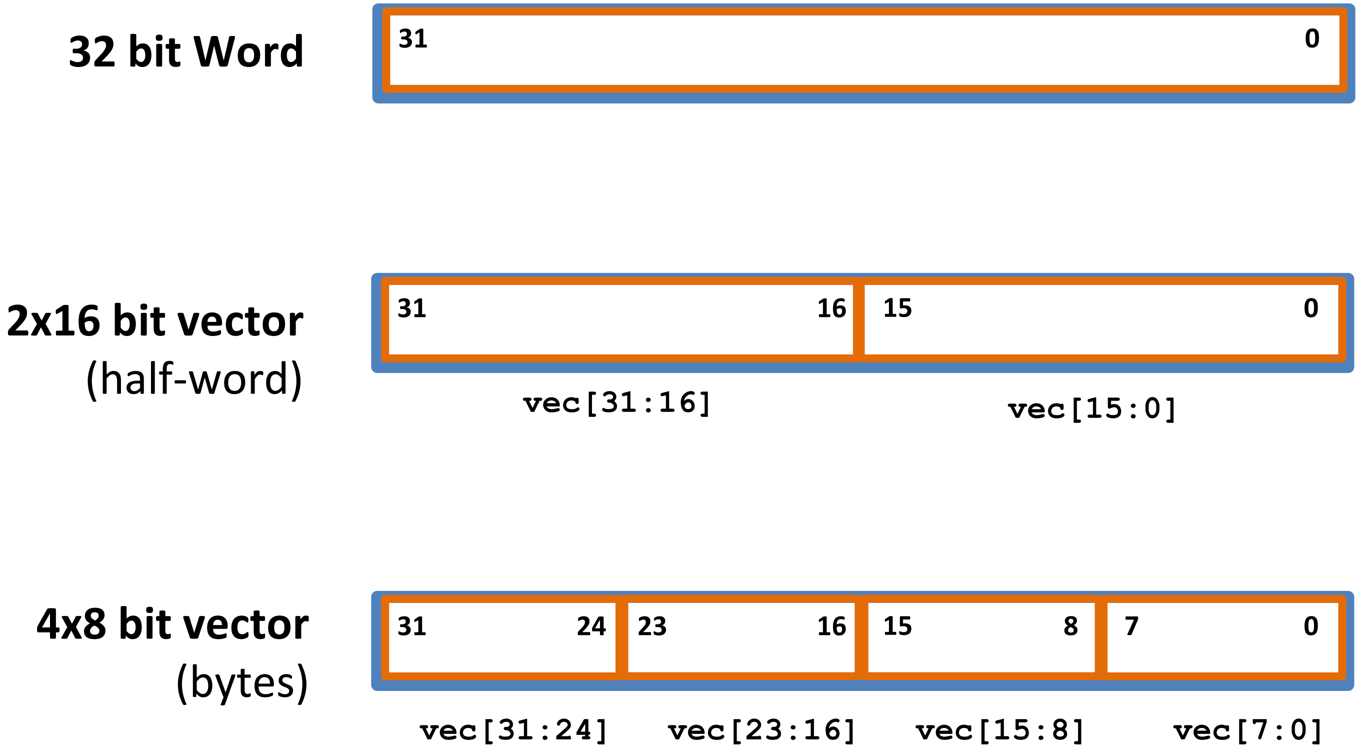
Check also the traces!

```
$ make clean all run CORES=8 runner_args="-trace=cluster/pe0/insn:log.txt"
```

TASK2.1: Vectorial Operations

Apply data-parallel processing on Vectorial Data

- **SIMD**: Single Instruction Multiple Data



32 bit Word

31 0

2x16 bit vector
(half-word)

31 16 | 15 0

`vec[31:16]` `vec[15:0]`

4x8 bit vector
(bytes)

31 24 | 23 16 | 15 8 | 7 0

`vec[31:24]` `vec[23:16]` `vec[15:8]` `vec[7:0]`

The content of a 32-bit register can be interpreted as a vector of **2x16 bit values** or **4x8 bit values**

Remember LAB02: we quantized the network to 8 bits.
--> This enables SIMD !

Example: Vectorial Add

Vectors are packed in the same integer register-file!

- The instructions encode how to interpret the content of the register

rs1	0x03	0x02	0x01	0x00
rs2	0x0D	0x0C	0x0B	0x0A

Vectorial Instructions of the Xpulp extension

add rD, rs1, rs2	rD = 0x03020100 + 0x0D0C0B0A

pv.add.h rD, rs1, rs2	rD[0] = 0x0100 + 0x0B0A
	rD[1] = 0x0302 + 0x0D0C

pv.add.b rD, rs1, rs2	rD[0] = 0x00 + 0x0A
	rD[1] = 0x01 + 0x0B
	rD[2] = 0x02 + 0x0C
	rD[3] = 0x03 + 0x0D

2x 16-bit ADD in one clock-cycles

4x 8-bit ADD in one clock-cycles

Leftovers (vectorization of fully connected)

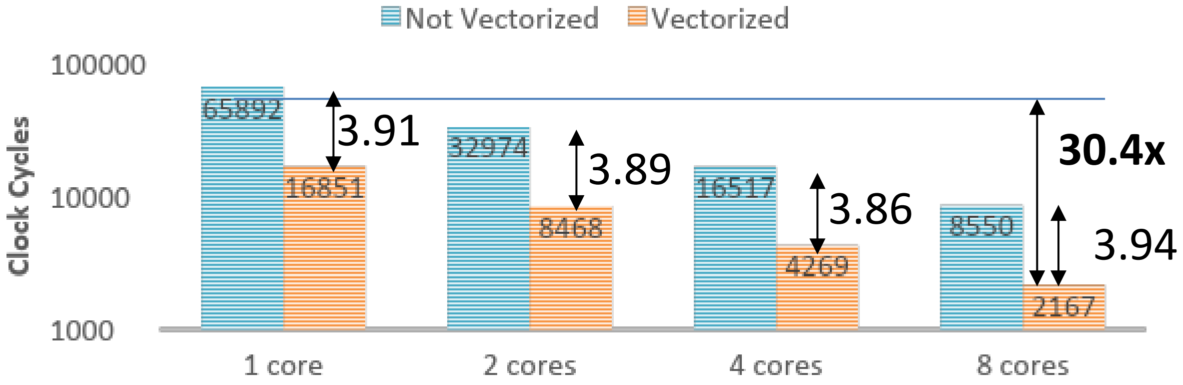
Replace this within the inner loop

```
v4u vecA;
v4s vecB;
// compute the vectorized dot products
for (int j=0; j<(dim_vec >> 2); j++) {
   vecA = *((v4u*)pA);
   vecB = *((v4s*)pB);
   sum = SumDotp4(vecA, vecB, sum);
   pA+=4;
   pB+=4;
}

// left over: handling the remaining input features
uint16_t col_cnt = dim_vec & 0x3;
while (col_cnt) {
  uint8_t inA = *pA;
  pA++;
  int8_t inB = *pB;
  pB++;
  sum += inA * inB;
  col_cnt--;
}
```

What if dim_vec is not divisible by 4? →Leftover

Handling the **left-over** (*dim_vec* %4 != 0)

Benchmarking the Vectorized Fully-Connected Layer



Combining parallelization and vectorization:

- ~8x on 8 cores
- ~4x thanks to vectorization (4x less iteration!)

With SIMD

# cores	#clk	#MAC /clk	#instr	# instr/ MAC
1				
2				
4				
8				

```
$ make dis
```

```
1c008bd8 <pulp_nn_linear_u8_i32_i8>:
......
1c008c40:     0068c0fb     lp.setup      x1,a7,1c008c4c <pulp_nn_linear_u8_i32_i8+0x74>
1c008c44:     004ea80b     p.lw          a6,4(t4!)
1c008c48:     004e278b     p.lw          a5,4(t3!)
1c008c4c:     a8f81357     pv.sdotusp.b  t1,a6,a5
.....
```

Similar to not-vectorized scheme:

- 3 instruction but 4 MAC now!

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

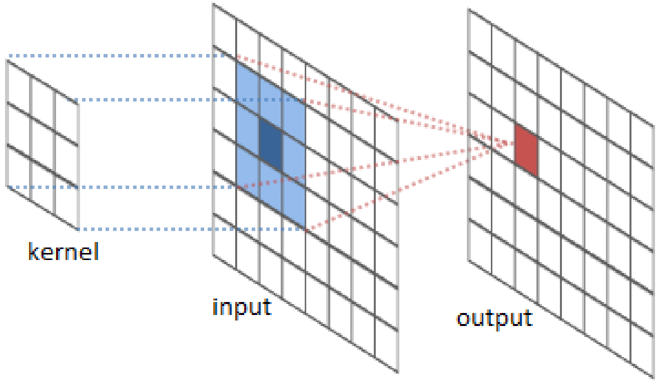
TASK3: Conv2D

Convolution Operation

Convolution: Basic Processing Kernel of Convolutional Neural Networks

$$\boldsymbol{y} = \boldsymbol{w} * \boldsymbol{x}$$

$$\boldsymbol{y}[i,j] = \sum_{u_i=0}^{F_i-1} \sum_{u_j=0}^{F_j-1} \boldsymbol{w}[u_i,u_j] * \boldsymbol{x}[i+u_i, j+u_j] \qquad (*)$$



(*) for math purists: "convolution" filters are technically ***cross-correlations*** *(convolutions with flipped weights in both dimensions) in most cases of interest for digital signal processing*

Convolution Operation

$$y[i,j] = \sum_{u_i=0}^{F_i-1} \sum_{u_j=0}^{F_j-1} w[u_i,u_j] * x[i+u_i, j+u_j]$$

Convolution: Basic Processing Kernel of Convolutional Neural Network

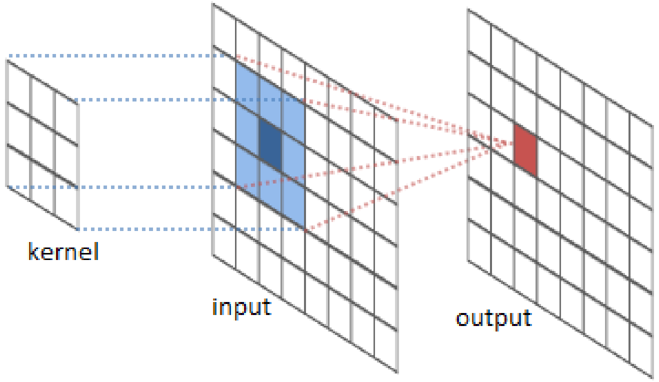
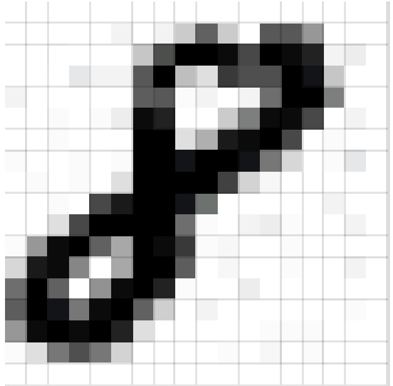


Image can be seen as Matrix



7	2	3	3	8
4	5	3	8	4
3	3	2	8	4
2	8	7	2	7
5	4	4	5	4

*

Kernel/Filter

1	0	-1
1	0	-1
1	0	-1

7x1+4x1+3x1+
2x0+5x0+3x0+
3x-1+3x-1+2x-1
= 6

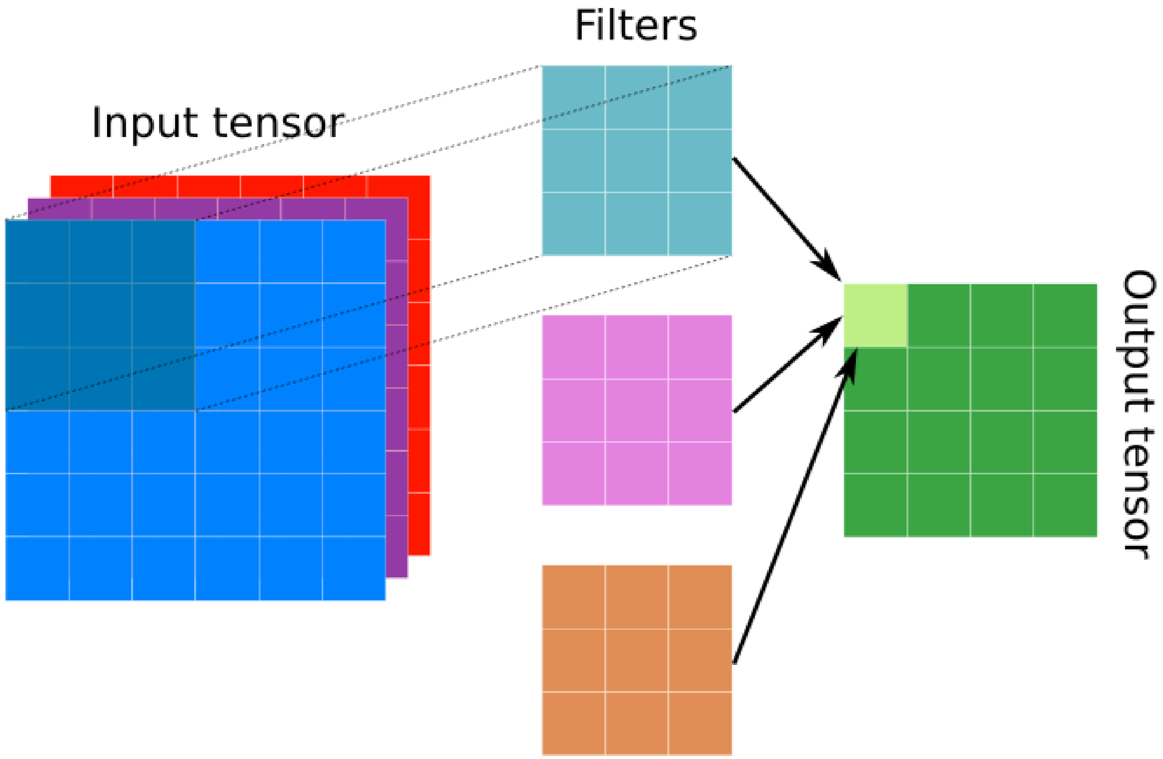
=

Feature Map

6		

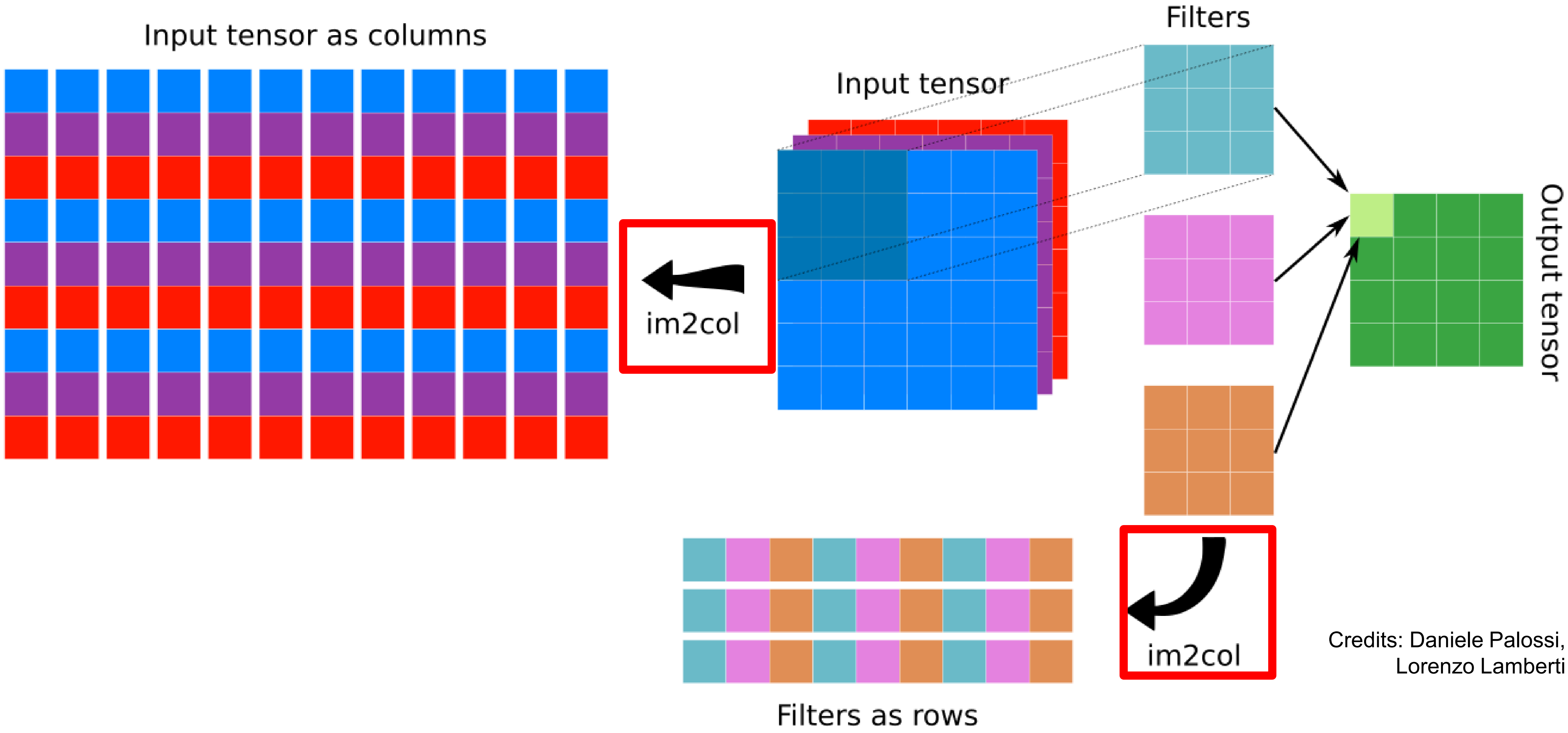
See example: http://bit.ly/hsdes21_conv2d

Convolution Operation: naive



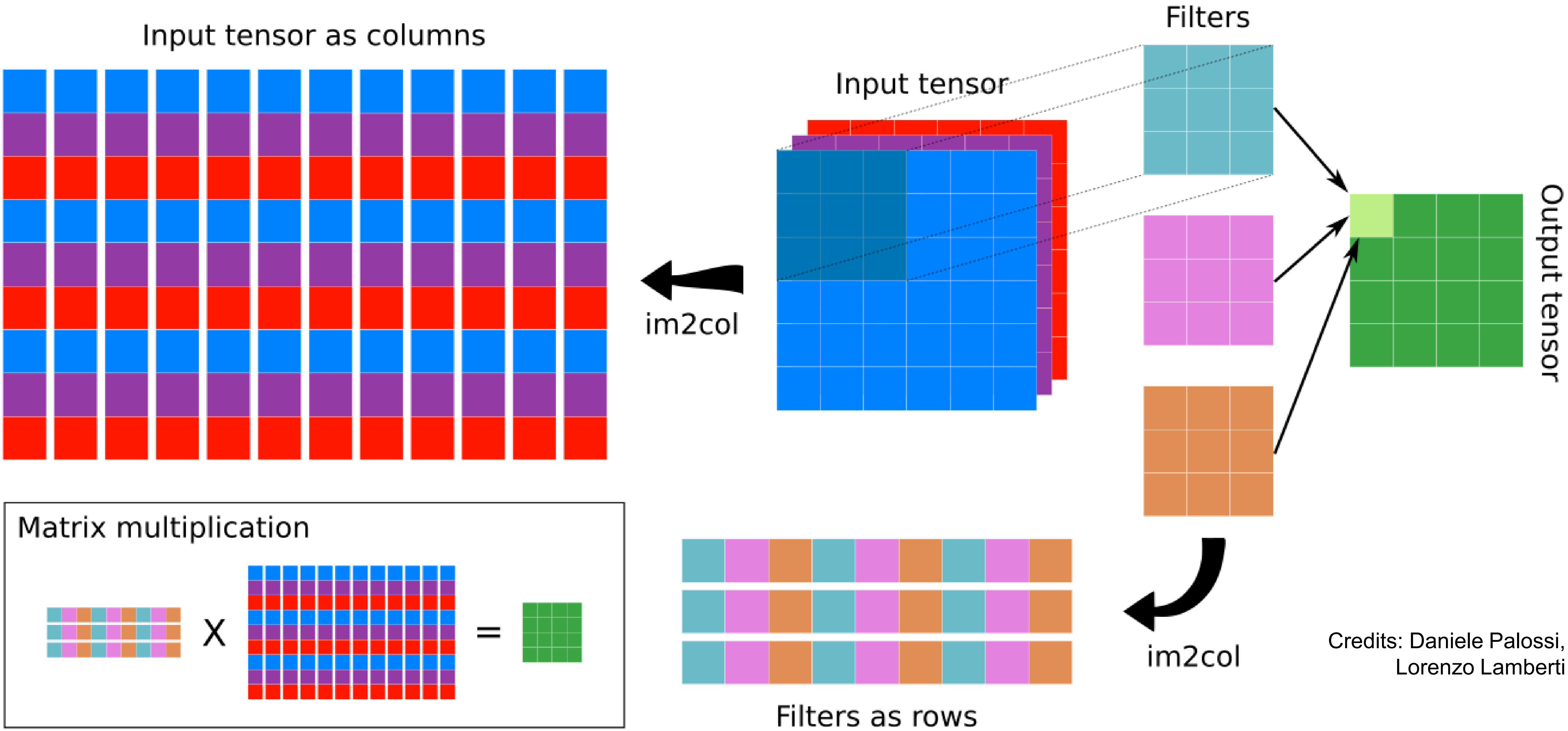
Credits: Daniele Palossi,
Lorenzo Lamberti

Convolution Operation: im2col and MatMul



Credits: Daniele Palossi, Lorenzo Lamberti

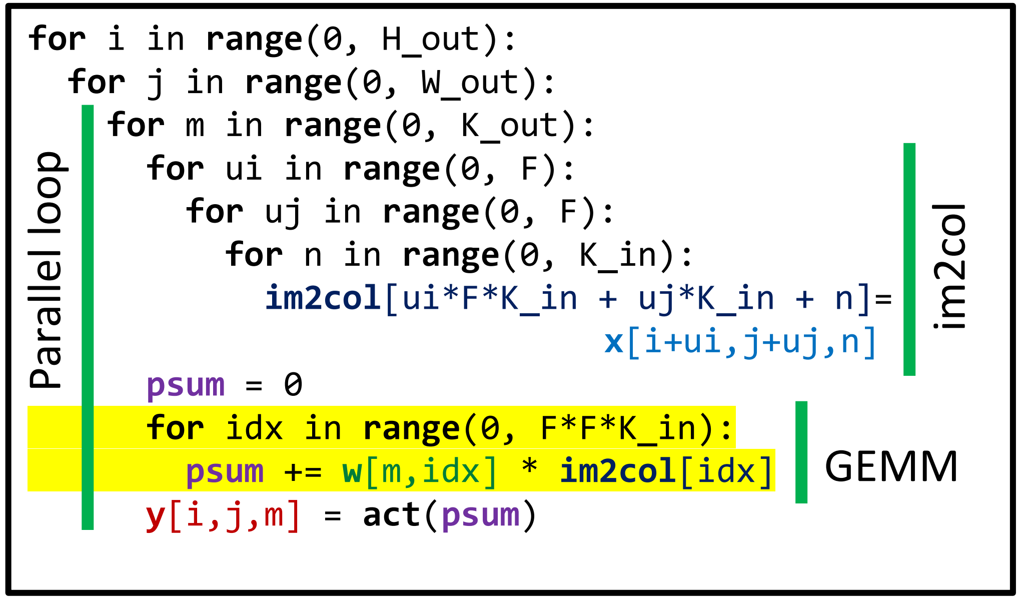
Convolution Operation: im2col and MatMul



Credits: Daniele Palossi,
Lorenzo Lamberti

DNN Conv2D Layer on PULP (PULP-NN)

```
$ cd conv2d/
$ make clean all run CORES=<1 to 8>
```



```
for i in range(0, H_out):
  for j in range(0, W_out):
    for m in range(0, K_out):
      for ui in range(0, F):
        for uj in range(0, F):
          for n in range(0, K_in):
            im2col[ui*F*K_in + uj*K_in + n]=
                              x[i+ui,j+uj,n]
      psum = 0
      for idx in range(0, F*F*K_in):
        psum += w[m,idx] * im2col[idx]
      y[i,j,m] = act(psum)
```

Implementation details:

- Standard **output stationary** loop nest
- **HWC** layout for activations, **CoHWCi** for weights
- 2 *im2col* buffer per core!!
- Unrolled MatMul

- Individuate parallelism, im2col and GEMM in the code
- Measure #clk_cyc and #instr if running conv2d on 1, 2, 4 and 8 cores
  - What is the percentage of workload of the GEMM?

Benchmarking the Conv2D Kernel

```
$ make dis
```

```
1c0091ec <pulp_nn_matmul_u8_i8>:
……
1c0092a8:     01c4c0fb     lp.setup       x1,s1,1c0092e0 <pulp_nn_matmul_u8_i8+0xf4>
1c0092ac:     0049a60b     p.lw           a2,4(s3!)
1c0092b0:     0049268b     p.lw           a3,4(s2!)
1c0092b4:     004ba88b     p.lw           a7,4(s7!)
1c0092b8:     004b280b     p.lw           a6,4(s6!)
1c0092bc:     004aa50b     p.lw           a0,4(s5!)
1c0092c0:     004a258b     p.lw           a1,4(s4!) # 4004 <pos_soc_event_callback+0x3bbc>
1c0092c4:     a9161f57     pv.sdotusp.b   t5,a2,a7
1c0092c8:     a9061ed7     pv.sdotusp.b   t4,a2,a6
1c0092cc:     a8a61e57     pv.sdotusp.b   t3,a2,a0
1c0092d0:     a8b61357     pv.sdotusp.b   t1,a2,a1
1c0092d4:     a9169457     pv.sdotusp.b   s0,a3,a7
1c0092d8:     a90693d7     pv.sdotusp.b   t2,a3,a6
1c0092dc:     a8a692d7     pv.sdotusp.b   t0,a3,a0
1c0092e0:     a8b69fd7     pv.sdotusp.b   t6,a3,a1
…..
```

From the assembly we expect 6 LD + 8 VMAC to compute 8x4 8 bit MAC

- 14 instr / 32 MAC = 0.437 instr / MAC

	clk	clk/MAC	Speed Up	instr	inst/MAC
1 core	2335846	0.49503	1	2297016	0.486801
2 cores	1169969	0.247949	1.996502	1148625	0.243425
4 cores	588218	0.12466	3.971055	575739	0.122015
8 cores	298860	0.063337	7.815854	289294	0.061309

From the measurements we see:

- 1 core almost as expected (workload dominated by the matmul inner loop)
- #instr ~ #cyc -> no stall!!
- Almost linear speedup!!

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEI – Università di Bologna

www.unibo.it