École d'ingénieurs de l'université de Nantes

2020/2021

Electronique et Technologies Numériques

# *PROGRAMMATION PARALLELE SUR CPU MULTICOEURS*

*Programmation vectorielle: SIMD-NEON*

*Parallélisation multitâches à mémoire partagée : OpenMP*

Hakim Saadane

Cinquième année

Premier réseau national des écoles d'ingénieurs polytechniques des universités

Site de la Chantrerie
Rue Christian Pauc – BP50609
44306 Nantes cedex 3 – France
Tél. +33 (0)2 40 68 32 00
Fax. +33 (0)2 40 68 32 32

Site de Gavy
Gavy Océanis – BP152
44603 St-Nazaire cedex – France
Tél. +33 (0)2 40 90 50 30
Fax. +33 (0)2 40 90 50 24

www.polytech.univ-nantes.fr

# Programmation parallèle sur CPU multicoeurs
"Programmation vectorielle: SIMD-NEON"

A.Saadane

Département Électronique et Technologies Numériques

## Summary

## Motivation

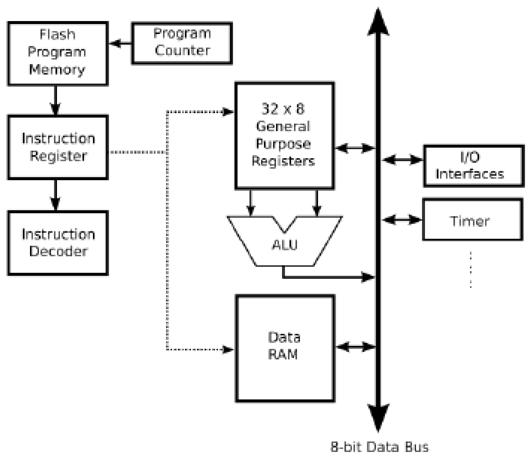
- Multimedia applications and products are proliferating :
  - Smartphones
  - iPad/tablets
  - Set top boxes
  - Servers and Networking infrastructure
- Systems currently use multiple processors :
  - Microcontrollers : arm Cortex M4,...
  - Application processors : arm V8, arm V8-A,...
  - GPU : nvidia,...
  - DSPs : AD, TI,...
  - SOCs : Intel,...
- The question : facing an application, what is the best choice ?

Generally speaking :

- **Microcontrollers : MCUs**
  - are integer math processors with an interrupt sub system,
  - are designed for simple math, and mostly to control other devices,
  - compact code that makes the most efficient use of the MCU architecture is essential.
- **DSPs**
  - are specialized microprocessors with architecture optimized for the operational needs of digital signal processing,
  - have special instructions such as multiply-accumulate in a single instruction (to speed up tasks),
  - can be designed as integer, fixed point or floating point processors,
  - are mainly used in specific audio and videos industries.
- **Application Processors**
  - High-end microcontrollers,
  - Clock speeds up to 2 GHz,
  - High level of integration : Multiple processor cores, Graphics coprocessor, Networking, USB, Security features.

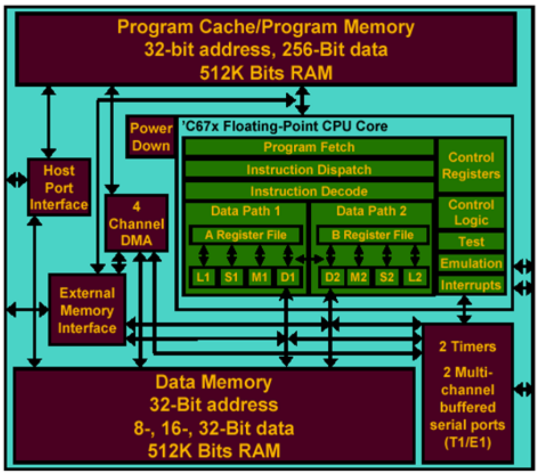
## MCU Architecture

- Single memory bus
- MAC takes 4 to 7 cycles
- Integer math with overflow
- No SIMD
- Floating-point hardware
- No circular and bit-reversed addressing



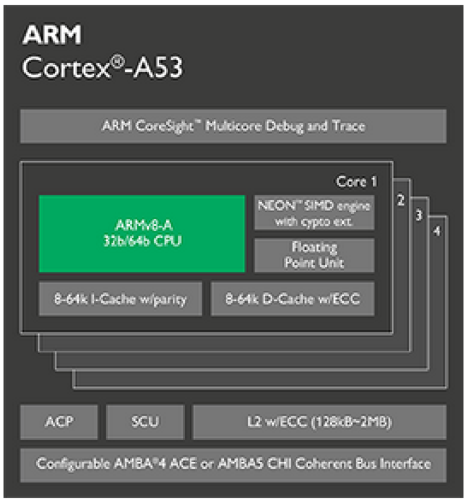
## DSP Architecture : TMS320C67XX

- Multiple memory buses (data, program)
- Single cycle multiply-accumulate (MAC)
- Loads and stores in parallel with computation
- Floating-point hardware
- Different addressing modes
- Multiple FU for parallel computation
- Large register files
- Multi-channel Buffered Serial Ports and Parallel Host Port Interface



## Application Processors Architecture : ARM Cortex-A53

- Low cost
- Large number of peripherals
- integrated flash memory
- Large and good driver support (USB, Ethernet,...)
- Operating systems
- Fully programmable from C
- Multiple processor cores
- Single cycle multiply-accumulate (MAC)
- Loads and stores in parallel with computation
- SIMD instructions for parallel computation


ARM
Cortex®-A53
ARM CoreSight™ Multicore Debug and Trace
Core 1
2
3
4
ARMv8-A 32b/64b CPU
NEON™ SIMD engine with cypto ext.
Floating Point Unit
8-64k I-Cache w/parity
8-64k D-Cache w/ECC
ACP
SCU
L2 w/ECC (128kB~2MB)
Configurable AMBA®4 ACE or AMBA5 CHI Coherent Bus Interface

## Expected improvements for DSP and Application processor

- Multimedia processing = DSPs or Application processors
- Expected improvements ( According to Paul Beckmann, DSP Concepts, LLC)

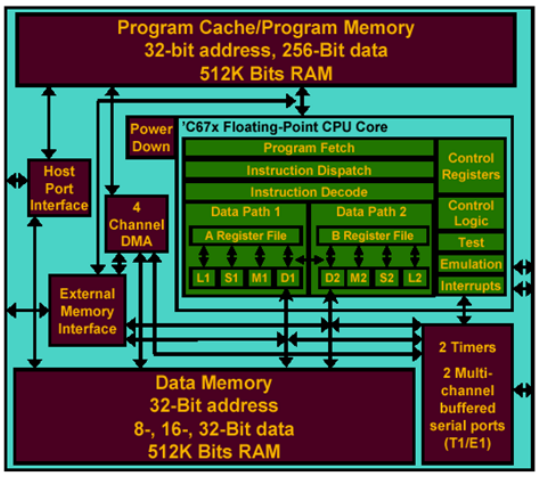
**Application processor**

- Better power consumption
- Audio specific peripherals
  - Serial ports
  - DAC and ADC
  - Flexible DMA controllers
- High performance arithmetic

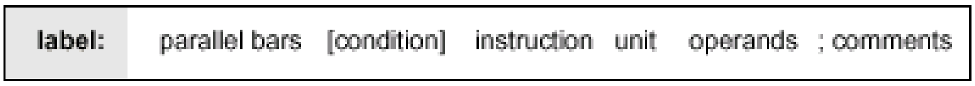
**DSP**

- Lower power sleep modes
- Lower cost
- Larger number of peripherals
- Integrated flash memory
- Good driver support : USB/Ethernet/CAN/Flash/etc
- Operating systems

- Program fetch Unit : to load a fetch packet (8 ins).
- Instruction Dispatch Unit : to send instructions to appropriate Functional Units.
- Instruction Decode Unit.
- 2 Data Paths : A et B each with four functional units (.L, .S, .M, .D).
- 32 32-bit registers.
- Control registers.
- Control logic.
- Test, emulation, and interrupt logic.



## Assembly code format



| | | | | | | |
|---|---|---|---|---|---|---|
| label: | parallel bars | [condition] | instruction | unit | operands | ; comments |

- label : is optional. If present, represents a specific address. Must be in the first column.
- || : if the instruction is being executed in parallel with the previous one.
- condition : optional field, used to make the associated instruction conditional.
- instruction : can be either an assembler directive or an instruction.
- unit : one of the eight CPU units.
- operands : All instructions require a destination operand. Most instructions require one or two operands sources.
- comments : must begin with a semicolon

## Examples of parallel and conditional operations

- Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name.

```
   [ B0 ] ADD .L1 A1, A2, A3
|| [! B0 ] ADD .L2 B1, B2, B3
```

- EP containing two instructions in parallel
  - The first ADD is conditional on B0 being nonzero.
  - The second ADD is conditional on B0 being zero (the character ! indicates the inverse of the condition).
  - The above instructions are mutually exclusive. Only one will execute.

- If they are scheduled in parallel, mutually exclusive instructions are constrained.

## Example of resource constraints

- No two instructions within the same execute packet can use the same resources.
- No two instructions can write to the same register during the same cycle.
- Example of constraints on using the Same Functional Unit

```
   ADD .S1 A0,A1,A2;            ADD .L1 A0,A1,A2;
|| SHR .S1 A3,15,A4;         || SHR .S1 A3,15,A4;
```

- Example of constraints on Cross Paths

```
   ADD .L1X A0,B1,A1;           ADD .L1X A0,B1,A1;
|| MPY .M1X A4,B4,A5;        || MPY .M2X B4,A4,B2;
```

- Example of constraints on Loads

```
   LDW .D1 *A0,A1;              LDW .D1 *A0,A1;
|| LDW .D2 *A2,B2;           || LDW .D2 *B0,B2;
```

## Introduction

- Software optimization is the process of manipulating software code to achieve two main goals :
  - Faster execution time.
  - Small code size.
- To implement efficient software, the programmer must be familiar with :
  - Processor architecture.
  - Programming language (C, assembly or linear assembly).
  - The code generation tools (compiler, assembler and linker).

## Assembly optimization

- Dot product for illustration

$$sum = \sum_{i=0}^{99} a[i]\, b[i]$$

- To implement this equation, we need :
  1. Load the coefficient a[i].
  2. Load the coefficient b[i].
  3. Multiply a[i] and b[i].
  4. Add (a[i] * b[i]) to the content of an accumulator.
  5. Repeat steps 1 to 4 99 times.
  6. Store the value in the accumulator to sum.

## full assembly code

```
MVK .S1 100,A1                 ; initialize loop counter to 100
ZERO .L1 A7                    ; clears the accumulator
LOOP
LDH .D1 *A4++, A2              ; load ai from memory
LDH .D1 *A3++, A5              ; load bi from memory
NOP 4                          ; allow delay slots of the LDH
MPY .M1 A2, A5, A6             (synchronization)
NOP                            ; ai * bi
ADD .L1 A6, A7, A7             ; delay slot for MPY
SUB .S1 A1, 1, A1              ; somme += (ai*bi)
[A1] B .S2 LOOP                ; decrement loop counter
NOP 5                          ; branch to loop
; Branch occurs here           ; delay slots for branch
```

- 16 cycles for each iteration plus two cycles to set up the loop counter and initialize the accumulator = 1602 cycles

## Assembly optimization

In order to optimize the code, we need to :

1. Use instructions in parallel.
2. Remove the NOPs.
3. Remove the loop overhead (remove SUB and B : loop unrolling).
4. Use word access or double-word access instead of byte or half-word access.

## Parallel assembly code and NOP reduction

```
   MVK .S1 100,A1                 ; initialize loop counter to 100
|| ZERO .L1 A7                    ; in parallel, zero out accumulator
LOOP
   LDH .D1 *A4++, A2              ; load ai from memory
|| LDH .D2 *B4++, B2              ; load bi from memory
   SUB .S1 A1, 1, A1              ; decrement loop counter
[A1] B .S2 LOOP                   ; branch to loop
   NOP 2                          ; delay slots for LDH
   MPY .M1X A2, B2, A6            ; ai * bi
   NOP                            ; delay slots for MPY
   ADD .L1 A6, A7, A7             ; somme += (ai*bi)
; Branch occurs here
```

- Eight cycles for each iteration plus one cycle to set up the loop counter and initialize the accumulator ; 100 iterations require 801 cycles.

- Because the loads of ai and bi do not depend on one another, both LDH instructions can execute in parallel (2 different FU).
- Because the MPY instruction now has one source operand from A and one from B, MPY uses the 1X cross path. By moving SUB and B, only 3 NOP are needed now (instead of 10).
- Rearranging the order of the instructions also improves the performance of the code. The SUB instruction can take the place of one of the NOP delay slots for the LDH instructions.
- Moving the B instruction after the SUB removes the need for the NOP 5 used at the end of the code.
- The branch now occurs immediately after the ADD instruction so that the MPY and ADD execute in parallel with the five delay slots required by the branch instruction

## SIMD Concept

- Power consumption = key factor in limiting the maximum processor operating speed
- Higher operating frequencies result in exponential increase in power.
- To overcome this issue, implement various micro-architectural enhancements to significantly increase the performance of a processor core, while operating at lower operating speeds with minimal impact in both power consumption.
- Towards more data parallelism and vector processing through the use of SIMD computation units (Single-Instruction/Multiple Data).
- Key concept behind SIMD = many sequential computations can be combined in parallel using specific machine instructions that operate on multiple data paths (registers/memory) simultaneously.

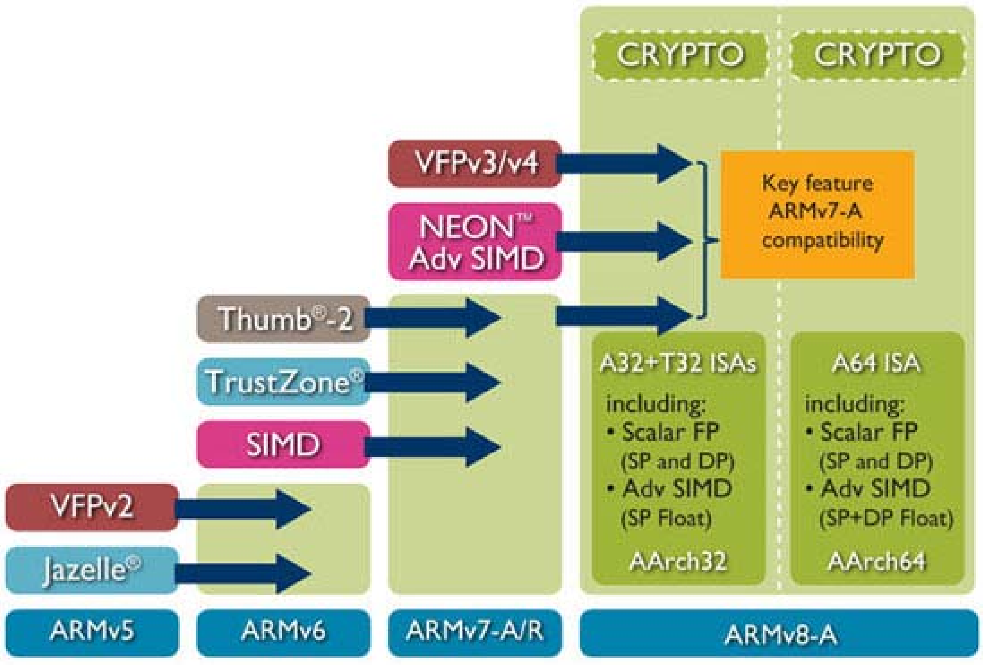
## Parallelism

- Ability to execute operations in parallel (more performant than clock speed improvements).
- 3 types of parallelism
  - SISD : Single Instruction Single Data (classical scalar processors).
  - SIMD : Single Instruction Multiple Data (Vector processors, data parallelism)
    - Well suited to DSP (Digital Signal Processing)
    - also suited to Application processor with SIMD engine
  - MIMD : Multiple Instruction Multiple Data (Multi-core processors, thread parallelism).

## ARM Architectures Profiles

- Application profile (ARMv7-A -> Cortex-A8)
  - Memory management support (MMU)
  - Optimized for rich operating systems
- Real-time profile (ARMv7-R -> Cortex-R4)
  - Protected memory (MPU)
  - Optimized for high-performance, hard real-time applications
- Microcontroller profile (ARMv7-M -> Cortex- M4)
  - Optimized for discrete processing and microcontroller

## Architectures


CRYPTO
CRYPTO
VFPv3/v4
NEON™ Adv SIMD
Key feature ARMv7-A compatibility
Thumb®-2
TrustZone®
SIMD
VFPv2
Jazelle®
A32+T32 ISAs including: • Scalar FP (SP and DP) • Adv SIMD (SP Float)
AArch32
A64 ISA including: • Scalar FP (SP and DP) • Adv SIMD (SP+DP Float)
AArch64
ARMv5
ARMv6
ARMv7-A/R
ARMv8-A

## Extensions

New extensions provided with every new version :

- Jazelle : Java hardware/software accelerator, technology for direct bytcode execution of Java.
- Vector Floating-Point : Coprocessor instruction set to support floating-point arithmetic.
- TrustZone : system-wide approach to security for a wide array of client and server computing platforms (include payment protection technology, digital rights management...)
- SIMD, NEON : the ARM SIMD 128 bit engine useful to make linear algebra operations.
- Crypto : extension of the SIMD support and operates on the vector register file. Provides instructions for the acceleration of encryption and decryption.

- AArch32 : the ARMv8-A 32-bit execution state. Uses 13 32-bit general purpose registers (R0-R12), a 32-bit program counter (PC), stack pointer (SP), and link register (LR). Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support. AArch32 execution state provides a choice of two instruction sets, A32 (ARM) and T32 (Thumb2). Operation in AArch32 state is compatible with ARMv7-A operation.
- T32 : 16-bit instructions decompressed transparently to full 32-bit ARM instructions in real time without performance loss. Thumb-2 technology made Thumb a mixed (32- and 16-bit) length instruction set.
- AArch64 : ARMv8-A 64-bit execution state. Uses 31 64-bit general purpose registers (R0-R30), and a 64-bit program counter (PC), stack pointer (SP), and exception link registers(ELR). Provides 32 128-bit registers for SIMD vector and scalar floating-point support (V0-V31). A64 instructions have a fixed length of 32 bits

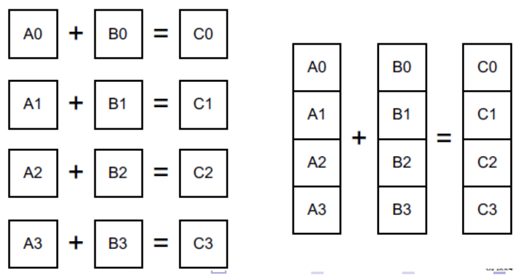
| Architectures | Core bit width | Cores designed by ARM |
|---|---|---|
| ARMv1 | 32 | ARM1 |
| ARMv2 | 32 | ARM2, ARM3 |
| ARMv3 | 32 | ARM6, ARM7 |
| ARMv4 | 32 | ARM8 |
| ARMv4T | 32 | ARM7TDMI, ARM9TDMI |
| ARMv5TE | 32 | ARM7EJ, ARM9E, ARM10E |
| ARMv6 | 32 | ARM11 |
| ARMv6-M | 32 | ARM Cortex-M0 et M1 |
| ARMv7-M | 32 | ARM Cortex M3 |
| ARMv7E-M | 32 | ARM Cortex-M4, M7 |
| ARMv7-R | 32 | ARM Cortex-R4, R5, R7, R8 |
| ARMv7-A | 32 | ARM Cortex-A5,A7,A8,A9,A12,A15,A17 |
| ARMv8-A | 32 | ARM Cortex-A32 |
| ARMv8-A | 64 | ARM Cortex-A35, A53, A57, A72 |

- Cores (mainly ARMv8-A) designed by third parties : Apple, AppliedMicro, Broadcom, Nvidia, Qualcomm Samsung

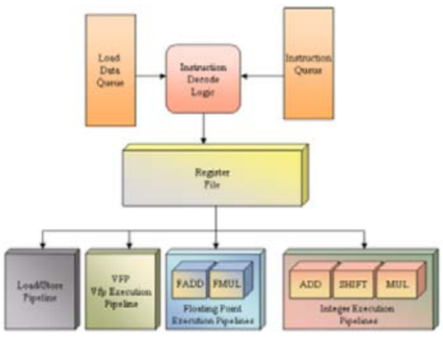
## What is NEON

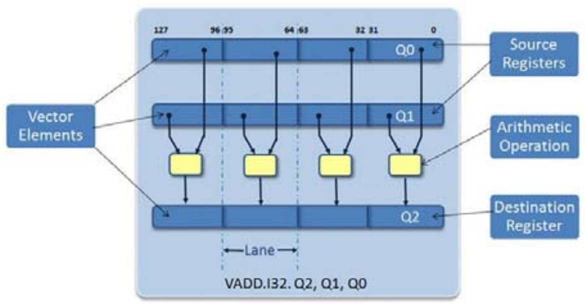
- NEON is general purpose SIMD engine.
- NEON units can accelerate multimedia and signal processing algorithms :
  - Block-based data processing (FFT, matrix multiplication, etc.)
  - Audio, image and video processing codecs (G72X, MPEG-X, H.264...)
  - 2D and 3D graphics
  - Gaming...
- NEON enables a **single instruction** to treat a register value as **multiple data** elements and to perform multiple, identical operations on those elements.

- Four separate additions using scalar operation = four add instructions.
- SIMD Parallel ADD needs only one instruction.



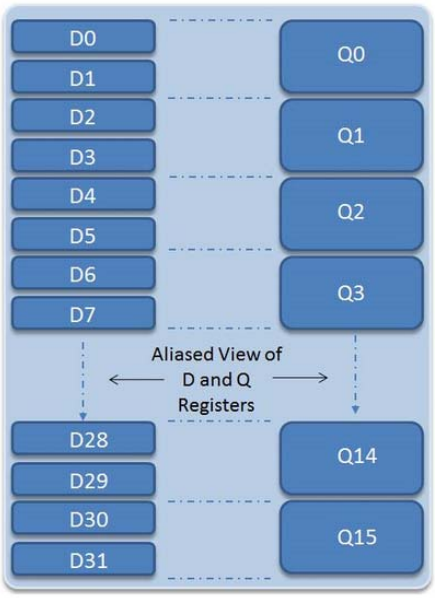
- NEON is integrated into the ARM core as a coprocessor.
- NEON operates with its own independent pipeline and register file.
- NEON shares registers with the vector floating point unit (VFP).
- NEON only works on vectors. VFP does not.
- Registers are considered as vectors of elements of the same data type :
- NEON instructions perform the same operation in all **lanes**.





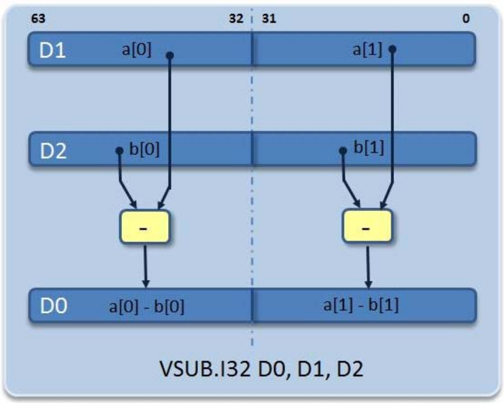
## NEON register bank

- Bank of a 256-byte (or 32x64-bit) register file.
- Distinct from core registers.
- Two explicitly aliased views :
  - either 32 x 64-bit **D** doubleword registers (D0 to D31)
  - or 16 x 128-bit **Q**, quadword registers (Q0-Q15).
- Vector instructions determine the appropriate register usage.



## NEON register bank (contd.)

- Operations can be done on 2, 4, 8 or 16 elements in parallel, depending on data type and register size.
- Operations on 32-bit integer allow 4 data elements in parallel.
- Operations on 16-bit integer allow 8 data elements in parallel.
- Operations on 8-bit integer allow 16 data elements in parallel.



## Data Types

Data type specifiers in NEON instructions consist of a letter that indicates the type of data and a number that indicates the width.

- Unsigned integer U8 U16 U32 U64.
- Signed integer S8 S16 S32 S64.
- Integer of unspecified type I8 I16 I32 I64.
- Floating-point number F16 F32.
- Polynomial over {0,1} P8

## Neon Instruction

General format of NEON instructions :

V{<mod>}<op>{<shape>}{<cond>}{.<dt>}(<dest>}, src1, src2

| | |
|---|---|
| mod | One of the modifiers (Q, H, D, R) |
| op | Operation (for example, ADD, SUB, MUL) |
| shape | Shape (L, W or N) |
| cond | Condition, used with IT instruction |
| .dt | Data type |
| dest | Destination |
| src1 | Source operand 1 |
| src2 | Source operand 2 |

## NEON Benefits

- Simple DSP algorithms : performance boost (4x-8x)
- Complex video codecs : performance boost about 60-150
- Better efficiency for memory access with wide registers
- Saves power (processor finish a task more quickly and enter sleep mode)

## Accessing NEON

1. Assembly : possible to get the absolute best performance
   - Needs experts
   - Enables experts writing the fastest/smallest possible implementation
   - Hard to comprehend, maintenance and development costs are high
   - Hardware implementation of Advanced SIMD, connection to the core logic and memory interface different within the ARM Cortex-A family
   - Consequently, NEON code that executes nearly optimal on one ARM CPU might not deliver the same performance on others.
2. Standard C : Implement in C and allow the C compiler to optimize as best as it can.
3. Tuned C : Hand optimize the code as best as possible while remaining in C. This involves loop unrolling, caching of variables, and using intrinsic functions (Performance close to assembly).

## Optimization using NEON from C

Three optimization methods :

1. Using NEON intrinsics :
   - function-like symbols translated to according assembler instructions depending on the target architecture,
   - are meant to replace the "inline assembler" mechanism,
   - looking like regular function calls,
   - NEON code easier to maintain than NEON assembler code,
   - New data type definitions that correspond to NEON registers (both D and Q registers),
   - Provide Low-level access to NEON instructions while letting the compiler doing the hard work (register allocation, scheduling...)

## Optimization using NEON from C

2. Using Compiler Automatic Vectorization :
   - throw arbitrary high-level code at the compiler (will make best use of all available coprocessor (neon) resources),
   - closely related to loop optimization,
   - enables loop transformation and vectorization if the optimization goal is execution time.
3. Using NEON optimized Libraries :
   - Ne10 : ARM experts optimized vector, matrix, and DSP functions.
   - FFmpeg : A complete, cross-platform solution to record, convert, and stream audio and video.
   - OpenCV Library aiming at real-time computer vision
   - ...

## Using NEON Intrinsics

**Compiling NEON intrinsics with GCC**

- Need to include header file "**arm_neon.h**"
- Use compiler options
  - -On. (default). Set the optimization levels.
  - -mcpu=cortex-aX. Set the processor type.
  - -mfpu=neon. Tell the compiler to generate NEON instructions
- Example : **gcc -mcpu=cortex-a9 -mfpu=neon -O3 -c test.c**

## Using NEON Intrinsics : Vector data types

**NEON vector data types** are defined according to :

**<type><size>x<number of lanes>_t**

- int32x4_t // vector of four 32-bit int elements
- uint16x8_t // vector of eight 16-bit uint elements
- float16x4_t // vector of four 16-bit float elements

**NEON array of vector types** are of the form :

**<type><size>x<number of lanes>x<length of array>_t**

- int8x8x2_t // array of two vectors with eight 8-bit elements

## Using NEON Intrinsics : NEON Intrinsics Instructions

Intrinsics use the following naming convention

**[opname][flags]_[type]**

- opname is a vector operator (instruction name),
- flag "q" means a quad word (128-bit) vectors,
- type = data type :
  - s16 single precision floating point
  - f32 double precision floating point
  - u8, u16 unsigned 8-bit, 16-bit integer

Examples :

- vmul_s16, multiplies two vectors of signed 16-bit values. compiles to VMUL.I16 d2, d0, d1.
- vaddl_u8, is a long add of two 64-bit vectors containing unsigned 8-bit values, resulting in a 128-bit vector of unsigned 16-bit values. compiles to VADDL.U8 q1, d0, d1.

## Using NEON Intrinsics : Some instructions : ADD

**Vector add : vaddq_<type>.**

Vr[i] :=Va[i]+Vb[i] Vr, Va, Vb have equal lane sizes

```
int32x2_t vadd_s32(int32x2_t a, int32x2_t b); // VADD.I32 d0,d0,d0
int8x16_t vaddq_s8(int8x16_t a, int8x16_t b); // VADD.I8 q0,q0,q0
uint8x16_t vaddq_u8(uint8x16_t a, uint8x16_t b); // VADD.I8 q0,q0,q0
```

**Example**

```
float32x4_t v1 = { 1.0, 2.0, 3.0, 4.0 }, v2 = { 1.0, 1.0, 1.0, 1.0 };
float32x4_t sum = vaddq_f32(v1, v2);
// => sum = { 2.0, 3.0, 4.0, 5.0 }
```

**Vector long add : vaddl_<type>.**

Vr[i] :=Va[i]+Vb[i] ; Va, Vb have equal lane sizes, result is a 128 bit vector of lanes that are twice the width.

```
int16x8_t vaddl_s8(int8x8_t a, int8x8_t b) ; // VADDL.S8 q0,d0,d0
int32x4_t vaddl_s16(int16x4_t a, int16x4_t b) ; // VADDL.S16 q0,d0,d0
uint16x8_t vaddl_u8(uint8x8_t a, uint8x8_t b) ; // VADDL.U8 q0,d0,d0
```

## Using NEON Intrinsics : Some instructions : MUL

**Vector multiply :**

**vmul{q}_<type>. Vr[i] := Va[i] * Vb[i]**

```
int16x4_t vmul_s16(int16x4_t a, int16x4_t b) ; // VMUL.I16 d0,d0,d0
uint8x8_t vmul_u8(uint8x8_t a, uint8x8_t b) ; // VMUL.I8 d0,d0,d0
int8x16_t vmulq_s8(int8x16_t a, int8x16_t b) ; // VMUL.I8 q0,q0,q0
```

**Example**

```
float32x4_t v1 = { 1.0, 2.0, 3.0, 4.0 }, v2 = { 1.0, 1.0, 1.0, 1.0 } ;
float32x4_t prod = vmulq_f32(v1, v2) ;
// => prod = { 1.0, 2.0, 3.0, 4.0 }
```

## Using NEON Intrinsics : Some instructions : MAC

**Vector multiply accumulate :**

**vmla{q}_<type>. Vr[i] := Va[i] + Vb[i] * Vc[i]**

```
int32x2_t vmla_s32(int32x2_t a, int32x2_t b, int32x2_t c);
  // VMLA.I32 d0,d0,d0
uint16x4_t vmla_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c);
  // VMLA.I16 d0,d0,d0
int8x16_t vmlaq_s8(int8x16_t a, int8x16_t b, int8x16_t c);
  // VMLA.I8 q0,q0,q0
```

**Example**

```
float32x4_t v1 = { 1.0, 2.0, 3.0, 4.0 }, v2 = { 2.0, 2.0, 2.0, 2.0 };
float32x4_t v3 = { 3.0, 3.0, 3.0, 3.0 };
float32x4_t acc = vmlaq_f32(v3, v1, v2); // S = A + B * C
// => acc =  5.0, 7.0, 9.0, 11.0
```

## Using NEON Intrinsics : Loads of a single vector or lane

**Load a single vector from memory**

- uint8x16_t vld1q_u8(__transfersize(16) uint8_t const * ptr);
  // VLD1.8 {d0, d1}, [r0]
- int32x4_t vld1q_s32(__transfersize(4) int32_t const * ptr);
  // VLD1.32 {d0, d1}, [r0]
- float32x4_t vld1q_f32(__transfersize(4) float32_t const * ptr);
  // VLD1.32 {d0, d1}, [r0]

**Example**

```
float values[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
float32x4_t v = vld1q_f32(values);
// => v = { 1.0, 2.0, 3.0, 4.0 }
```

## Using NEON Intrinsics : Loads of a single vector or lane

### Load a single lane from memory

- uint16x8_t vld1q_lane_u16(__transfersize(1) uint16_t const * ptr, uint16x8_t vec, __constrange(0,7) int lane) ;
  // VLD1.16 {d0[0]}, [r0]
- int32x4_t vld1q_lane_s32(__transfersize(1) int32_t const * ptr, int32x4_t vec, __constrange(0,3) int lane) ;
  // VLD1.32 {d0[0]}, [r0]
- float16x8_t vld1q_lane_f16(__transfersize(1) __fp16 const * ptr, float16x8_t vec, __constrange(0,7) int lane) ;
  // VLD1.16 {d0[0]}, [r0]

## Using NEON Intrinsics : Store a single vector or lane

### Store a single vector into memory

- void vst1q_u8(__transfersize(16) uint8_t * ptr, uint8x16_t val) ;
  // VST1.8 {d0, d1}, [r0] ;
- void vst1q_s8(__transfersize(16) int8_t * ptr, int8x16_t val) ;
  // VST1.8 {d0, d1}, [r0]
- void vst1q_f32(__transfersize(4) float32_t * ptr, float32x4_t val) ;
  // VST1.32 {d0, d1}, [r0]

### Example

```
float32x4_t v = { 1.0, 2.0, 3.0, 4.0 } ;
float values[5] = new float[5] ;
vst1q_f32(values, v) ;
// => values = { 1.0, 2.0, 3.0, 4.0, #undef }
```

## Using NEON Intrinsics : Store a single vector or lane

**Store a lane of a vector into memory**

- void vst1q_lane_u8(__transfersize(1) uint8_t * ptr, uint8x16_t val, __constrange(0,15) int lane); // VST1.8 {d0[0]}, [r0]
- void vst1q_lane_s16(__transfersize(1) int16_t * ptr, int16x8_t val, __constrange(0,7) int lane); // VST1.16 {d0[0]}, [r0]
- void vst1q_lane_f16(__transfersize(1) __fp16 * ptr, float16x8_t val, __constrange(0,7) int lane); // VST1.16 d0[0], [r0]

## Using NEON Intrinsics : Tools

- Declaring a variable

  uint32x2_t vec64a, vec64b; // create two D-register variables
- Using constants

  uint8x8_t start_value = vdup_n_u8(0); // duplicates a constant into each element of a vector
- Moving results back to normal C variables

  result = vget_lane_u32(vec64a, 0); // extract lane 0
- Accessing two D-registers of a Q-register

  vec64a = vget_low_u32(vec128); // split 128 bit vector
  vec64b = vget_high_u32(vec128); // into 2x 64 bit vectors
- Casting NEON variables between different Ttypes

  uint8x8_t byteval;
  uint32x2_t wordval;
  byteval = vreinterpret_u8_u32(wordval);

## Vector Addition : First example

- Vector addition C code

```c
void add_scalar(uint8_t * A, uint8_t * B, uint8_t * C){
    for(int i=0; i<16; i++){
        C[i] = A[i] + B[i];
    }
}
```

- Vector addition NEON intrinsics code (code assembleur pour illustrer)

```c
void add_neon(uint8_t * A, uint8_t * B, uint8_t *C){
        //Setup a couple vectors to hold our data
        uint8x16_t vectorA, vectorB vectorC;

        //Load our data into the vector's register
        vectorA = vld1q_u8(A);
        vectorB = vld1q_u8(B);

        //Add A and B together
        vectorC = vaddq_u8(vectorA, vectorB);
}
```

  - uint8x16_t is a vector type containing 16 8-bit uints in an array.
  - vld1q_u8 loads 8-bit uints into a vector.
  - vaddq_u8 adds two vectors made of 8bit uints together all at once.
  - if tested, NEON function isn't faster (two loads).

## Vector Addition : second example

- Vector addition C code

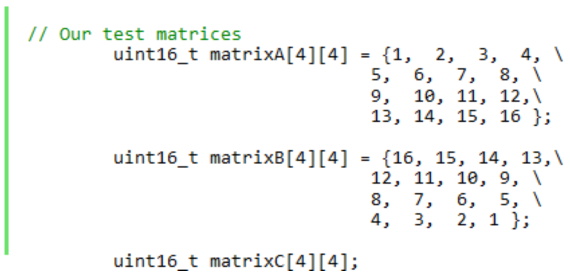
```c
void NeonTest(int * x, int * y, int * z)
{
int i;
for(i=0;i<200;i++) {
z[i] = x[i] + y[i];
}
}
```

- Vector addition NEON intrinsics code (code assembleur pour illustrer)

```c
#include "arm_neon.h"
void intrinsics(uint32_t *x, uint32_t *y, uint32_t *z)
{
  int i;
  uint32x4_t x4,y4; // These 128 bit registers will contain 4 values from the x array and 4 values from the y array
  uint32x4_t z4;    // This 128 bit register will contain the 4 results from the add intrinsic
  uint32_t *ptra = x; // pointer to the x array data
  uint32_t *ptrb = y; // pointer to the y array data
  uint32_t *ptrz = z; // pointer to the z array data
```

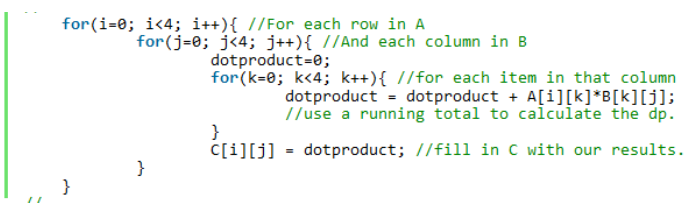
```c
  for(i=0; i < 200/4; i++)
  {
    x4 = vld1q_u32(ptra);  // intrinsic to load x4 with 4 values from x
    y4 = vld1q_u32(ptrb);  // intrinsic to load y4
    z4=vaddq_u32(x4,y4);   // intrinsic to add z4=x4+y4
    vst1q_u32(ptrz, z4);   // store the 4 results to z
    ptra+=4; // increment pointers
    ptrb+=4;
    ptrz+=4;
  }
```

## Matrix Multiplication



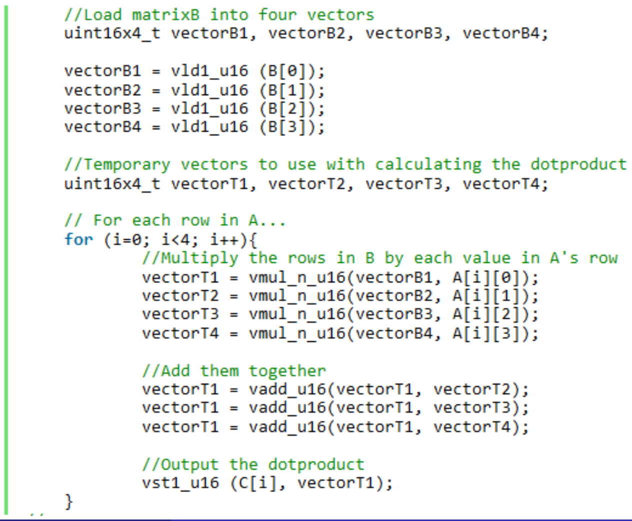
```
// Our test matrices
        uint16_t matrixA[4][4] = {1,  2,  3,  4, \
                                  5,  6,  7,  8, \
                                  9,  10, 11, 12,\
                                  13, 14, 15, 16 };

        uint16_t matrixB[4][4] = {16, 15, 14, 13,\
                                  12, 11, 10, 9, \
                                  8,  7,  6,  5, \
                                  4,  3,  2,  1 };

        uint16_t matrixC[4][4];
```

- C Code



```
   for(i=0; i<4; i++){ //For each row in A
          for(j=0; j<4; j++){ //And each column in B
                 dotproduct=0;
                 for(k=0; k<4; k++){ //for each item in that column
                        dotproduct = dotproduct + A[i][k]*B[k][j];
                        //use a running total to calculate the dp.
                 }
                 C[i][j] = dotproduct; //fill in C with our results.
          }
   }
```

- Neon code



```
//Load matrixB into four vectors
uint16x4_t vectorB1, vectorB2, vectorB3, vectorB4;

vectorB1 = vld1_u16 (B[0]);
vectorB2 = vld1_u16 (B[1]);
vectorB3 = vld1_u16 (B[2]);
vectorB4 = vld1_u16 (B[3]);

//Temporary vectors to use with calculating the dotproduct
uint16x4_t vectorT1, vectorT2, vectorT3, vectorT4;

// For each row in A...
for (i=0; i<4; i++){
        //Multiply the rows in B by each value in A's row
        vectorT1 = vmul_n_u16(vectorB1, A[i][0]);
        vectorT2 = vmul_n_u16(vectorB2, A[i][1]);
        vectorT3 = vmul_n_u16(vectorB3, A[i][2]);
        vectorT4 = vmul_n_u16(vectorB4, A[i][3]);

        //Add them together
        vectorT1 = vadd_u16(vectorT1, vectorT2);
        vectorT1 = vadd_u16(vectorT1, vectorT3);
        vectorT1 = vadd_u16(vectorT1, vectorT4);

        //Output the dotproduct
        vst1_u16 (C[i], vectorT1);
}
```

## Using Compiler Auto-vectorization

- The easiest way to optimize for NEON.
- A speed optimization where computations are performed on several values simultaneously by converting scalar implementation to a vector implementation.
- A vector is a set of scalar data items, all of the same type, stored in memory.
- Vector processing occurs when arithmetic and logical operation are applied to vectors.
- The conversion from scalar processing to vector code by the compiler is called vectorization.

## Using Compiler Auto-vectorization

- Two steps for this conversion :
  - the compiler will first analyze the source code and determine whether certain loops can be mapped to a vector algorithm,
  - the compiler will then transform the scalar loop into a sequence of vector operations which perform arithmetic computations on a block of elements.
- Advantages of automatic vectorization on C or C++ source code
  - access to high NEON performance without writing assembly code or using intrinsics
  - source code remains portable between different tools and target platforms
  - have to give the compiler additional hints about where this is safe and optimal

## Using Compiler Auto-vectorization : optimization levels

Optimization levels set using the command line option -On :

- -O0. (default). No optimization is performed (clearest view for source level debugging but the lowest level of performance)
- -O1. Enables the most common forms of optimization that do not require decisions regarding size or speed (faster than -O0).
- -O2. Enables further optimizations, such as instruction scheduling.
- -O3. Enables more aggressive optimizations. Typically increases speed at the expense of image size. Enables -ftree-vectorize (compiler attempts to generate NEON code).
- -Os. Optimizations minimize the size of the image even at the expense of speed.

## Using Compiler Auto-vectorization : optimization options

Options used to tell the compiler to generate NEON instructions :

- -std=c99. The C99 standard introduces some new features that can be used for NEON optimization.
- -mcpu=cortex-a9. Specifies the name of the target ARM processor to determine appropriate instructions.
- -mfpu=neon to use NEON coprocessor.
- -ftree-vectorize. Performs loop vectorization on trees (enabled at level '-O3').
- -mvectorize-with-neon-quad. Vectorizes for quad-word for better code performance.

Example :
gcc -mcpu=cortex-a53 -mfpu=neon -ftree-vectorize
-mvectorize-with-neon-quad -c test.c

## Tuning source code for vectorization

- C and C++ standards do not provide syntax that specifies parallel behavior.
- C code must be tuned to provide additional hints to the compiler
- Recommendation techniques for tuning code for Neon :
  - Write code that is straightforward and parallel (help the compiler to convert code to NEON assembly)
  - Indicate the number of loop iterations
  - Tune loops for better vectorization (see below)
  - Use the 'restrict' keyword
  - Use suitable data types (smallest data type is always the best choice, NEON can process twice as many 8-bit values as 16-bit values.)

## Tuning source code for vectorization

**Tune Loops**

- loops Short, simple loops work best
- Number of loop iterations should be a power of 2
- Avoid breaks / loop-carried dependencies (result of one iteration affected by the result of previous iterations, no vectorization)
- Avoid conditions inside loops (hard for the compiler to vectorize loops containing conditional sequences)

**Use the 'restrict' keyword**

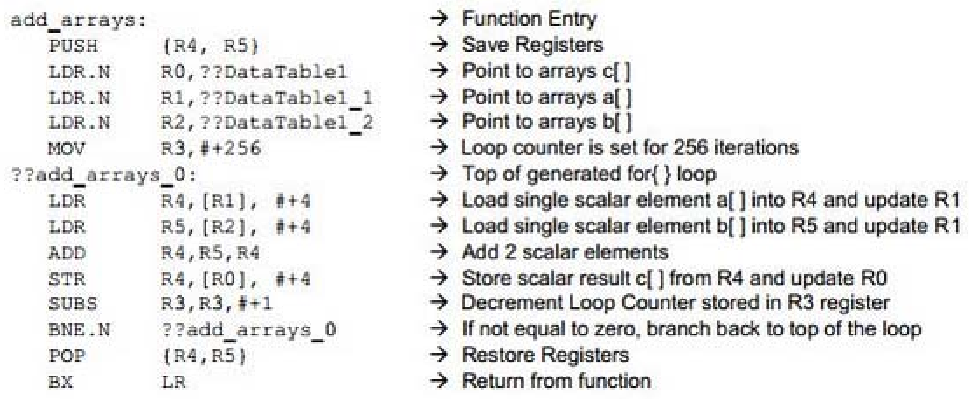
- Access arrays by simple indexing vectorizes better than using pointers
- Avoid indirect addressing (doesn't vectorize well)
- Use the __restrict keyword to tell the compiler that pointers do not reference overlapping areas of memory (Otherwise compiler might assume that pointers refer to the same location et will not vectorize)

## Example : Vector Addition

- C code

```c
int a[256], b[256], c[256];
void add_arrays()
{
    int i;

    for (i = 0; i < 256; ++i)
    {
        c[i] = a[i] + b[i];
    }
}
```

- Assembly code generated by the compiler

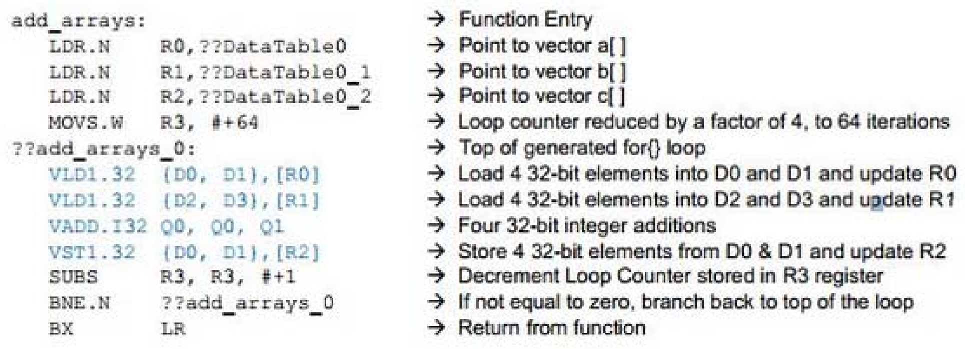


```
add_arrays:                          → Function Entry
    PUSH     {R4, R5}                → Save Registers
    LDR.N    R0,??DataTable1         → Point to arrays c[ ]
    LDR.N    R1,??DataTable1_1       → Point to arrays a[ ]
    LDR.N    R2,??DataTable1_2       → Point to arrays b[ ]
    MOV      R3,#+256                → Loop counter is set for 256 iterations
??add_arrays_0:                      → Top of generated for{ } loop
    LDR      R4,[R1], #+4            → Load single scalar element a[ ] into R4 and update R1
    LDR      R5,[R2], #+4            → Load single scalar element b[ ] into R5 and update R1
    ADD      R4,R5,R4                → Add 2 scalar elements
    STR      R4,[R0], #+4            → Store scalar result c[ ] from R4 and update R0
    SUBS     R3,R3,#+1               → Decrement Loop Counter stored in R3 register
    BNE.N    ??add_arrays_0          → If not equal to zero, branch back to top of the loop
    POP      {R4,R5}                 → Restore Registers
    BX       LR                      → Return from function
```

- Total of 1541 cycles : 256 iterations x 6 instructions + 5 cycles

## Vector Addition : vectorization enabled

- Assembly code



```
add_arrays:                                → Function Entry
    LDR.N      R0,??DataTable0             → Point to vector a[ ]
    LDR.N      R1,??DataTable0_1           → Point to vector b[ ]
    LDR.N      R2,??DataTable0_2           → Point to vector c[ ]
    MOVS.W     R3, #+64                    → Loop counter reduced by a factor of 4, to 64 iterations
??add_arrays_0:                            → Top of generated for{} loop
    VLD1.32    {D0, D1},[R0]               → Load 4 32-bit elements into D0 and D1 and update R0
    VLD1.32    {D2, D3},[R1]               → Load 4 32-bit elements into D2 and D3 and update R1
    VADD.I32   Q0, Q0, Q1                  → Four 32-bit integer additions
    VST1.32    {D0, D1},[R2]               → Store 4 32-bit elements from D0 & D1 and update R2
    SUBS       R3, R3, #+1                 → Decrement Loop Counter stored in R3 register
    BNE.N      ??add_arrays_0              → If not equal to zero, branch back to top of the loop
    BX         LR                          → Return from function
```

- Rx registers have been replaced by the NEON's Dx and Qx registers.
- The vector instructions will load, add and store four array elements simultaneously.
- The loop counter was reduced from 256 iterations (without vectorization) to 64 (with vectorization).
- Total of 388 cycles : 64 iterations x 6 instructions + 4 cycles

# Programmation parallèle sur CPU multicoeurs

## "Parallélisation multitâches à mémoire partagée: OpenMP"

A.Saadane

Département Électronique et Technologies Numériques

## Summary

## L'environnement OpenMP

- OpenMP est une API (Application Programming Interface) utilisée pour faciliter la programmation parallèle avec mémoire partagée.
- OpenMP est une notation qui peut être ajoutée à un langage séquentiel (en C, C++...) pour décrire comment le travail peut être partagé entre les threads (tâches) à exécuter sur différents processeurs.
- OpenMP suit le modèle fork/Join :
  - Les programmes OpenMP commencent avec un seul thread : le thread maître (Thread#0).
  - Au début d'une région parallèle, le thread maître crée une équipe (groupe) de threads "travailleurs" parallèles (fork).
  - les codes d'un bloc parallèle sont exécutés en parallèle par chaque thread.
  - A la fin de la région parallèle, tous les threads se synchronisent et rejoignent le thread maître (Join).

OpenMP est constitué de 3 composantes :

- les directives de compilation : utilisées pour la création de région parallèles, la répartition des blocs de code sur les threads, la distribution des boucles itératives entre les threads, la sérialisation des sections de code et la synchronisation du travail entre threads.
- les fonctions d'exécution : utilisées pour fixer et se renseigner sur le nombre de threads, récupérer l'identifiant unique du thread, se renseigner si une région est parallèle et à quel niveau.....
- les variables d'environnement : utilisées pour fixer le nombre de threads, spécifier comment les itérations de boucle sont divisées, relier les threads aux processeurs...

- Les directives OpenMP permettent de spécifier au compilateur quelles instructions sont à exécuter en parallèle et comment distribuer ces instructions entre les threads qui exécuteront le code.
- Une directive OpenMP est une instruction qui n'est reconnue que par les compilateurs OpenMP. (Cette directive est assimilée à un commentaire par tout autre compilateur).
- Les prototypes des différentes fonctions sont dans le fichier omp.h qu'il faut inclure préalablement dans le fichier source.
- Les directives commencent toujours par **#pragma omp**
- La forme générale d'une directive est

**#pragma omp nom-directive [clause[[,] clause]...]**

## Syntaxe : Création de Threads

**#pragma parallel**

- Les threads communiquent moyennant des variables partagées.
- Une synchronisation est nécessaire entre les threads pour éviter les conflits entre données (changement de données quand les threads sont cadencés (organisés) différemment).
- Le nombre de threads peut être fixé moyennant :
  - la variable environnementale OMP_NUM_THREADS,
  - la fonction (d'exécution) omp_set_num_threads(n).

## Création de Threads (suite)

- La création de threads se fait par la directive "parallel"

```
#pragma omp parallel num_threads(4)
{% Le code à l'intérieur de cette région s'exécute en parallèle
printf("Hello !\n");
}
```

- Cette directive crée un groupe (équipe) de 4 threads ici, et chaque thread exécute le même code.
- Le résultat de l'exemple ci-dessus affichera 4 fois "Hello" avec un retour à la ligne.
- Le nombre de threads peut être spécifié avant la directive

```
omp_set_num_threads(4);
#pragma omp parallel
{...code...
}
```

## Création de Threads (suite et fin)

- Les fonctions permettant d'avoir l'information liée aux threads sont :
  - **int omp_get_num_procs()** %pour connaître le nombre de coeurs disponibles.
  - **int omp_get_thread_num()** %pour récupérer le numéro du thread.
  - **int omp_get_num_threads()** %pour avoir le nombre total de threads.
  - **double omp_get_wtime()** % pour avoir le temps en seconde.

## Création de boucles

**#pragma omp for**

- La directive "for" divise la boucle de sorte que chaque thread du bloc courant gère une partie différente de la boucle.

```
#pragma omp for
for(int n=0; n<10; n++)
{ printf(" %d", n);
}
```

- La sortie de cette boucle affichera les chiffres de 0 à 9 mais dans un ordre arbitraire.
- En interne, cette boucle est équivalente au code suivant :

```
int thread_courant = omp_get_thread_num(),
num_threads = omp_get_num_threads();
int start = (thread_courant ) * 10 / num_threads;
int end = (thread_courant+1) * 10 / num_threads;
for(int n=start; n<end; n++)
printf(" %d", n);
```

## Création de boucles (suite)

**Remarques :**

- "#pragma omp for" ne délègue des parties de la boucle qu'aux threads du bloc (équipe) courant.
- Pour créer un nouveau bloc de threads, il y a besoin d'utiliser la directive "parallel".
- En langage C, la variable de boucle (n dans l'exemple précédent) est nécessairement une variable privée (voir section ci-dessous) :

  *int n ;*
  *#pragma omp for private(n)*
  *for(n=0 ; n<10 ; n++) printf(" %d", n) ;*

## Différences entre "parallel", "parallel for" et "for"

- Un bloc (équipe) est un groupe de threads qui s'exécute selon le modèle fork/join. Au lancement du programme, le bloc consiste en un seul thread. La directive "parallel" divise le bloc courant en un nouveau bloc de threads (travailleurs) pour la durée du prochain bloc, après quoi les différents threads refusionnent (join) en un seul.
- "for" divise le travail de la boucle for entre les threads du bloc actuel.
- "for" ne crée pas de threads, il se contente de diviser le travail entre les threads du bloc en cours d'exécution.
- "parallel for" est un raccourci pour les deux commandes : "parallel" crée un nouveau bloc et "for" partage ce bloc pour gérer les différentes parties de la boucle.

## Ordonnancement de boucle

- L'ordonnancement de boucle est utilisé pour contrôler la manière dont les itérations de boucle sont distribuées entre les threads.
- 2 paramètres sont utilisés pour ce contrôle : kind et chunksize.
- kind spécifie le type d'ordonnancement : "static" (par défaut) ou "dynamic" et chunksize spécifie le nombre d'itérations à traiter par thread.

  ```
  #pragma omp for schedule(static)
  for(int n=0; n<10; n++) printf(" %d", n);
  printf(".\n");
  ```

- Dans le mode "static" et une fois dans la boucle, chaque thread décide indépendamment quel chunk (nombre d'itérations contiguës) de la boucle il va traiter.

## Ordonnancement de boucle (suite)

- Dans le mode "dynamic"

  ```
  #pragma omp for schedule(dynamic)
  for(int n=0; n<10; n++) printf(" %d", n);
  printf(".\n");
  ```

- il n'y a pas d'ordre prévisible dans lequel les éléments de boucle sont affectés à différents threads.
- Chaque thread demande à la bibliothèque d'exécution OpenMP un nombre d'itération, qu'il traite.
- Une fois terminé il redemande un autre nombre d'itération et ainsi de suite. Ceci est particulièrement utile lorsque les différentes itérations de la boucle nécessitent des temps différents d'exécution.

## Ordonnancement de boucle (suite et fin)

- Le chunksize peut également être spécifié pour diminuer le nombre d'appels à la bibliothèque d'exécution :

  ```
  #pragma omp for schedule(dynamic, 3)
  for(int n=0; n<10; n++) printf(" %d", n);
  printf(".\n");
  ```

- Dans cet exemple, chaque thread demande un nombre d'itérations, exécute 3 itérations de la boucle, puis en demande un autre, et ainsi de suite.

## Sections

**#pragma omp section**

- La directive "sections" permet de spécifier que tel travail et tel autre peuvent être exécutés en parallèle. *#pragma omp sections*

  ```
  {
  { Work1(); }
  #pragma omp section
  { Work2();
  Work3(); }
  #pragma omp section
  { Work4(); }
  }
  ```
- Ce code spécifie que Work1, Work2+Work3 et Work4 peuvent s'exécuter en parallèle.
- Work2 et Work3 s'exécutent séquentiellement.

## Sécurité des threads (exclusion mutuelle)

Quelques outils sont utilisés pour gérer correctement les exclusions mutuelles.

- Atomicity : signifie que quelque chose est inséparable ; il doit se produire complètement ou pas du tout et aucun thread ne peut intervenir durant son exécution.

  ```
  #pragma omp atomic
  count += a
  ```
  atomic est généralement utilisée pour la mise à jour des compteurs ou de toute autre variable accessible par différents threads.

- Directive "critical" : restreint l'exécution de l'instruction/bloc associé à un seul thread à la fois. Deux threads ne peuvent exécuter une directive "critical" du même nom dans le même temps.

  ```
  #pragma omp critical(dataupdate)
  {
  datastructure.reorganize();
  }
  ...
  #pragma omp critical(dataupdate)
  {
  datastructure.reorganize_again();
  }
  ```
  Dans cet exemple, seule une des sections "critical" appelée "dataupdate" peut être exécutée à tout moment et seulement par un seul thread à cet instant.

## Clauses "private" et "shared"

A l'intérieur d'un bloc, OpenMP offre la possibilité de déclarer les variables comme étant privées ou partagées.

- shared (liste) : Toutes les variables dans "liste" sont partagées et chaque thread a accès à ces variables.
- private(liste) : toutes les variables dans "liste" sont privées au thread et aucun autre thread ne peut y accéder.

```
#pragma omp parallel for
shared(A,x,w)
for(int n=0; n<50; n++)
   x[n] = A*sin(w*n);
```

Dans cet exemple n est une variable privée (indice de boucle). A noter que toute variable privée est initialement non définie. Ce qui peut générer des erreurs :

```
int x = 5;
#pragma omp parallel for private(x)
for(int n=0; n<50; n++)     % la valeur de x est ici non définie
```

## Clause "firstprivate"

La clause "firstprivate" est un cas spécial de la clause "private"'. Elle initialise chaque copie privée avec la valeur correspondante du thread maître.

```
void fprive()
{
int tmp = 0;
#pragma omp for firstprivate(tmp)
for (int j = 0; j < 1000; ++j)
tmp += j;
printf("%d\n", tmp)
}
```

La valeur affichée par cet exemple est 0 puisque chaque thread récupère son propre "tmp" avec la valeur initiale égale à zéro.

## Clause "lastprivate"

La clause "lastprivate" passe (change) la valeur de a "private", à partir de la dernière itération, à une variable globale.

```
void lprive()
{
int tmp = 0;
#pragma omp parallel for firstprivate(tmp)\
lastprivate(tmp)
for (int j = 0; j < 1000; ++j)
tmp += j;
printf("%d\n", tmp)
}
```

La valeur affichée par cet exemple est 999 puisque c'est la valeur de "tmp" à la dernière itération séquentielle.

## Clause "default"

La clause "default(none)" est surtout utilisée pour vérifier que toutes les variables sont déclarées comme "shared" ou "private" pour partager ou pas les données entre les threads.

```
int a, b=0;
#pragma omp parallel default(none) shared(b)
{
b += a;
}
```

Ce code ne compilera pas et exigera de spécifier explicitement si la variable a est partagée ou privée. Une solution peut être de déclarer que toutes les variables par défaut sont partagées avec la clause "default(shared)".

## Clause "reduction"

- La clause "reduction" est un mix entre les clauses private, shared, et atomic.
- Cette clause permet d'accumuler une variable partagée sans la clause atomic, mais le type d'accumulation doit être spécifié.
- La syntaxe de la clause est *reduction(operator :list)* où "list" est la liste des variables où l'opérateur doit s'appliquer.
- Operator est un des opérateurs arithmétiques (+, -, *, ˆ) ou logiques (|, ||, &, &&)

L'exemple de calcul du factoriel d'un nombre

```
int factorial(int number)
{
int fac = 1;
#pragma omp parallel for
for(int n=2; n<=number; ++n)
{
#pragma omp atomic
fac *= n;
}
return fac;
}
```

L'exemple de calcul du factoriel d'un nombre peut être réécrit comme suit

*int factorial(int number)*
*{*
*int fac = 1 ;*
*#pragma omp parallel for reduction(* :fac)*
*for(int n=2 ; n<=number ; ++n)*
*fac *= n ;*
*return fac ; }*

Un autre exemple pour illustrer la clause "reduction" est le calcul du produit scalaire
#pragma omp parallel for reduction(+ :sum)
for (i=0 ; i < N ; i++)
sum = sum + (a[i] * b[i]) ;
Ici la clause "reduction" spécifie 2 choses :

- Chaque thread de la région parallèle prend une copie privée de sum (sum étant partagée) initialisée avec l'élément neutre (0 pour la somme).
- Chaque thread s'occupe d'un certain nombre d'itérations pour lesquelles il fait le produit et calcule la somme.
- A la fin, la variable originale sum est mise à jour en combinant toutes les copies privées de tous les threads. La somme étant associative, le résultat final est le même que pour une exécution séquentielle.

Sans la clause "reduction", le cumul est séquentiel (attente q'un thread ait fini avant que le suivant ne commence : performance réduite).

## Imbrication de boucle

Un exemple d'imbrication de boucle :

```
#pragma omp parallel for
for(int y=0; y<25; ++y)
{
#pragma omp parallel for
for(int x=0; x<80; ++x)
{
tick(x,y);
}
}
```

- La lecture de ce code laisse penser que les 2 boucles sont parallélisées. En fait, seule la boucle extérieure l'est.
- La deuxième boucle est séquentielle et se comporte comme si #pragma est ignorée.
- Au début de la boucle interne, OpenMP détecte qu'un groupe (équipe) existe déjà (crée par le premier #pragma et au lieu de créer un nouveau groupe de threads, il va créer un nouveau groupe composé

Ce problème peut être résolu par l'utilisation de la clause "collapse" dans la directive for.

```
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
for(int x=0; x<80; ++x)
{
tick(x,y);
}
```

Le nombre spécifié dans la clause collapse est le nombre de boucles imbriquées sujettes au partage du travail.