ELEC-H-473
Microprocessor architecture
Data Parallelism

Lecture 06

Dragomir MILOJEVIC
dragomir.milojevic@ulb.ac.be

Previously on ELEC-H-473

- **Instruction Level Parallelism** — execute different instruction phases in parallel to improve execution throughput
- Many techniques proposed to improve **instruction bandwidth** jeopardised with **hazards**
- CPUs are faster then memories, we need to add **memory hierarchy (i.e. cache)** to keep-up with fast cores and many different management technique to improve **data throughput**
- More processing implies more execution pipelines : **super-scalar** architectures they can really execute multiple instructions (same stage) at the same time
- But this is not all the parallelism that we have : binary data is sometimes more parallel that we think ...

Today

1. Motivation for data parallelism

2. Parallel processing :
   old classification that is still up-to-date

3. SIMD with Intel x86

4. How to use SIMD with x86 ?

5. Examples of SIMD programming with Intel CPUs

1. Data parallelism : motivation

On ALU and data types

- ALUs are designed to be efficient on computations for most commonly used **data types** in applications they target
- Common data types are integers and floating point with **various bit-widths**
- The ALU complexity depends on the operand width, so the **CPU architects trade-off : ALU perf. vs. area (cost)**
- If we need very simple computations most of the time, 16-bit ALU may be more then enough ...
- In that case & if the application needs bigger operands occasionally, the ALU could support that at the expense of performance : the computation on more complex operands is decomposed into a sequence of smaller operations (typically done in few cycles)

GPP ALUs

- Most General Purpose CPUs have ALUs that work to satisfy the **largest data type possible** and therefore satisfy us all
  - ✓ This is possible thanks to better & better IC integration (scaling)
  - ✓ Doubling of the minimum data-size over the years : from 4 to 64 bits
- The inverse problem : **What happens when a program uses smaller data types?**
- Example of the 32 bit ALU doing operations on unsigned integer coded with 8 bits :

```
Min : 0x00          0x000000B5
Max : 0xFF        + 0x00000011
```

Unused bits !

How inefficient it is ?

- Consider the following program :

```
unsigned char A, B, C;
int i;
for (i = 0 ; i < 1000000; i++) {
C[i] = A[i] + B[i];
}
```

- Assume execution on 64 bit ALU
- The usage of the ALU is only 12.5%
  - ✓ How did I come up with 12.5% figure?

When using ALUs designed for wider data types in computations using smaller data types, there is a significant loss in computational efficiency …

Solution : pack more data to use all 64 bits

```
unsigned char A, B, C;
int i;
for (i = 0 ; i < 1000000; i+=4) {
C[i+0] = A[i+0] + B[i+0];
C[i+1] = A[i+1] + B[i+1];
C[i+2] = A[i+2] + B[i+2];
C[i+3] = A[i+3] + B[i+3];
}
```

4 X better !

	i+3	i+2	i+1	i
0x	10	BB	A0	B5
+ 0x	30	FF	AB	11

Pack data
Use the same ALU &
Stop propagating that carry !

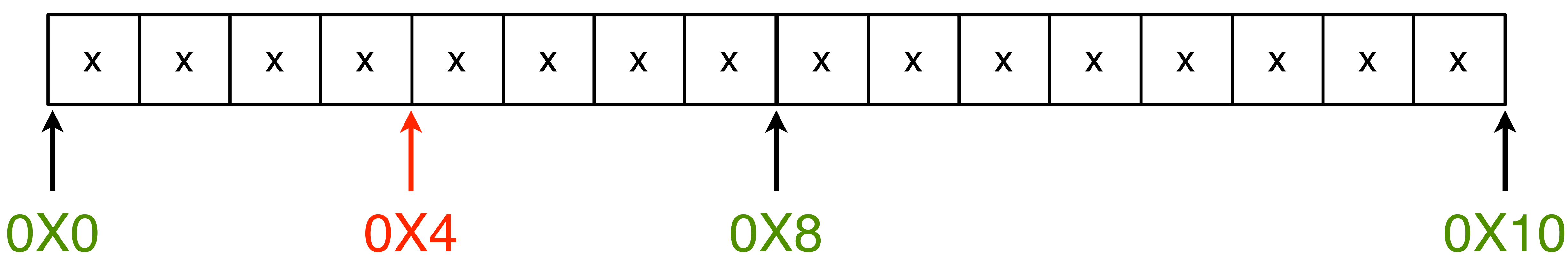
This solution has another benefits ...

- Reads **sets of data** in registers, rather then individual elements
  - ✓ If a bus transfer between central-memory/cache and cache/CPU, is 64 bits, this is the optimal data size to be transferred, not 8 bits
- Sets of data are nothing else then **vectors**
  →why this is often called: **vector processing**
- If we have smaller data types **AND** need to do vector operations we could modify ALU so that if can work in vector mode
- This is also known as **sub-word parallelism**
- But memory access have some issues that make them sub-optimal in some cases ...

Memory alignment

- A memory address A is aligned if A mod n = 0, where n is the width of the accessed data in bytes
- When a memory address is misaligned, the value A mod n determines the offset from alignment
- Alignment plays an crucial role for the efficiency of the memory access : **aligned access are much faster then non-aligned** (lot of design/architecture efforts have been spent over the years to overcome this, but this is still holds)

0X0, 0X8, 0x10 are all 64 bit aligned at 64 boundary
0x4 is not



Impact of the memory alignment

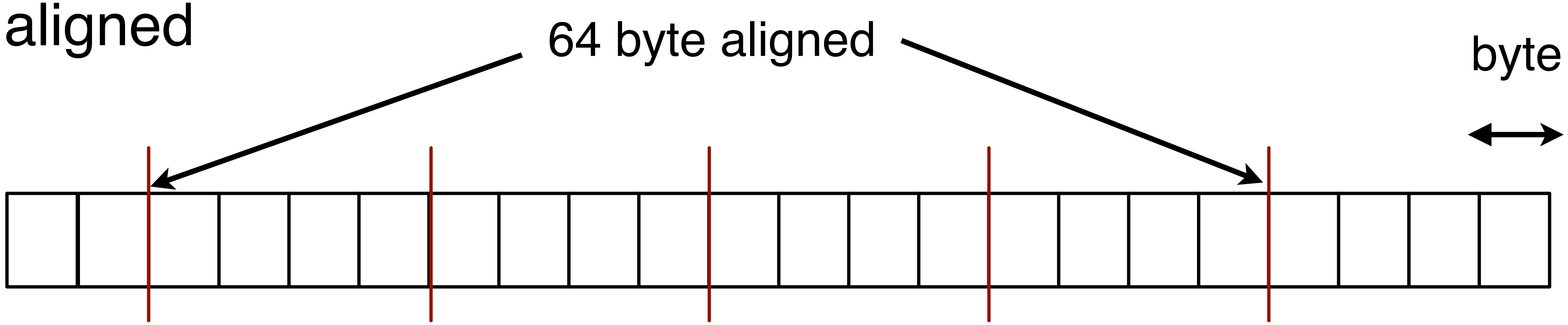
```
unsigned char *a, *b, *c;
int i;
for (i = 0; i< N; i++)
    a[i] = b[i] + c[i];
```

```
unsigned char *a, *b, *c;
int i;
for (i = 0; i< N; i++)
    a[i] = b[i+1] + c[i+2];
```

- Consider the two programs :
  - ✓ Data types are bytes
  - ✓ If you assume 8 byte alignment
  - ✓ `i+1` , or `i+2` index for the arrays is **non-aligned** !
- If you keep the order as such → **performance loss**
- Many solutions to solve this: realign everything (especially if N is big), loose some of the processing, etc.

Yet another issue : cache line split

- Assume 64 byte wide cache line and 16 byte alignment
- If you access data at the 64 byte boundary, you will use well your cache because the cache read at the boundary will use all 64 bytes
- If not **more then one cache line** needs to be read to allow the access to the data — **cache line split**
- This will occur on all accesses, even those that are 16 byte aligned



- Cache line splits are blockers : some architectures introduce very specific data movement instructions to avoid this

Classification

Sequential processing

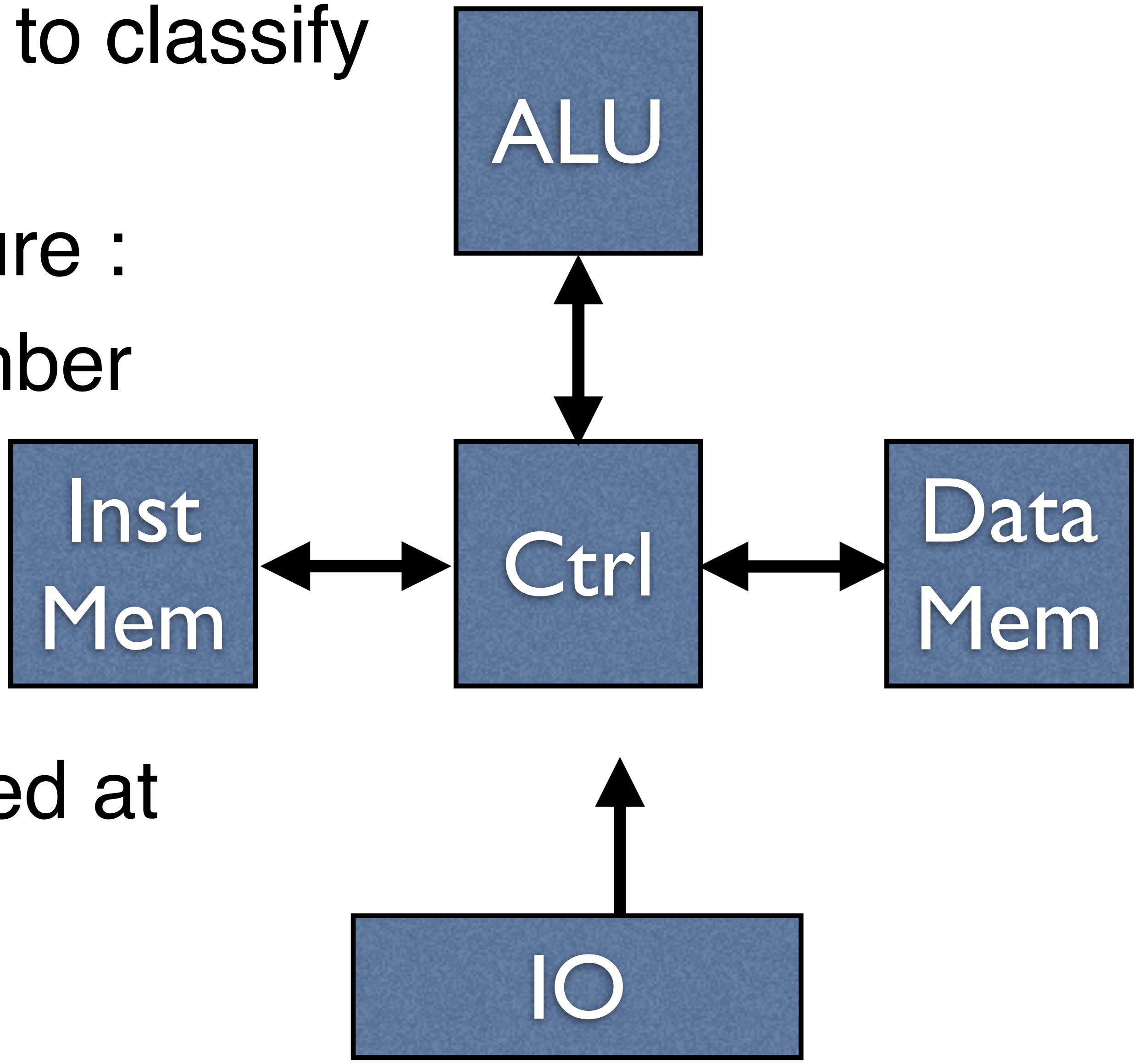
- In our computer architecture model we assumed that the ALU exhibits **bit-level parallelism** :
  - ✓ Any arithmetic/logic operation, sequential by nature (e.g. addition) is considered to be done in parallel, i.e. it is computed in one clock cycle
    - ❖ In reality this operation could be also decomposed in a certain sequence of operations, but let's stop at this abstraction level
- To keep the complexity of the ALU low, the number of operands is in general limited to 2, but in "nature" the applications need more then 2 operands … or are more complex
- Complex, or computations on more operands, are done as sequence of simple computations, thus we speak about **sequential processing** (note that the sequence here is not the same as in logic circuit language)

Parallel processing

- As opposed to sequential processing, if we do some computations concurrently (in the same time) we can speak of **parallel processing (**not the first time here)
- Because the execution of the instruction is a lengthy process, we decomposed it in sub-set to enable pipeline execution : **Instruction-Level Parallelism**
- This was first parallel execution that we saw, other then ALU operating in parallel on the bits of the operands word
- We also added multiple execution pipelines to increase the parallelism and allow data independent instructions to be computed in parallel — **super-scalar**
- So, there are many **parallel** things going on, let's put some order in it ...

Options for parallel computations

- **Sequential vs. parallel** : how to classify computer architectures?
- Look at the Harvard architecture :
- Relationship between the number of occurrences of :
  - ✓ **instructions**
  - ✓ **data**
- How many of these are handled at each execution cycle :
  - ✓ **single**
  - ✓ **multiple**
- Doing simple combinatorics over these two give us the classification of computing systems according to Flynn



Flynn's classification

- **Single Instruction Single Data** (SISD) — at each clock cycle only one instruction works on **one pair of operands**; this is what we already have in our model
- **Single Instruction Multiple Data** (SIMD) — at each clock cycle one instruction works on **sets of operands**; this is what we already have in our model
→ it is the vector processing we were talking about …
- **Multiple Instruction Single Data** (MISD) — at each clock cycle set of instruction work on the same data; very specific, but used
- **Multiple Instruction Multiple Data** (MIMD) — this is multi-CPU, multi-program environment
(Can you tell what would be a MIMD computer today?)

SIMD concept

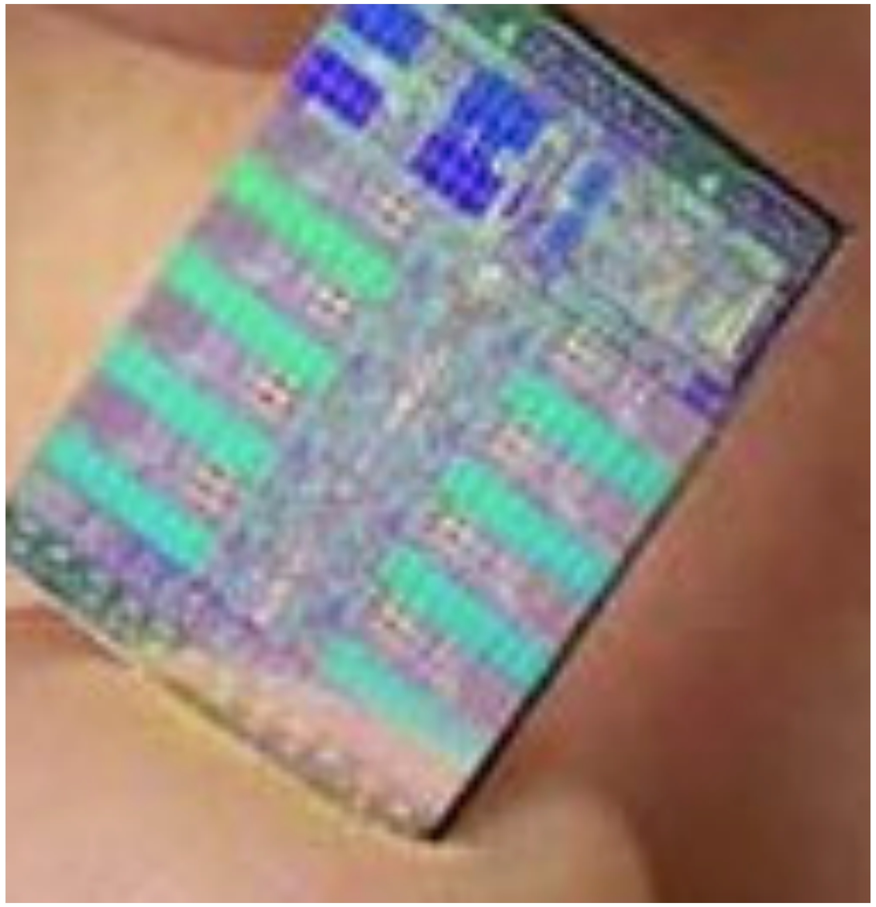
- Heavily used in ’60, ’70 and ’80 to build super-computers
  - ✓ CRAY — you can see one in CDC @ULB
  - ✓ Thinking Machines — Connection Machines CM1 → CM5
- But super-computers moved, into another direction (load of general purpose CPUs)
- … and SIMD moved into processors, to increase the ALU generality (this was mainly driven by appearance of multimedia end of ’90)

Why SIMD moved into CPU architectures ?

- Multimedia : audio, image, video … and combinations that are in the engineering vocabulary coupled to **Signal Processing applications**
- **Typical applications** :
  - ✓ Speech compression algorithms and filters & speech recognition algorithms
  - ✓ Video display and capture routines
  - ✓ Rendering routines & 3D graphics (geometry)
  - ✓ Image and video processing algorithms
  - ✓ Spatial (3D) audio
  - ✓ Physical modeling (graphics, CAD)
  - ✓ Encryption algorithms, complex arithmetics
- **Signal** : 1, 2 or 3D → **huge arrays of small data types** → perfect candidates for SIMD !!!

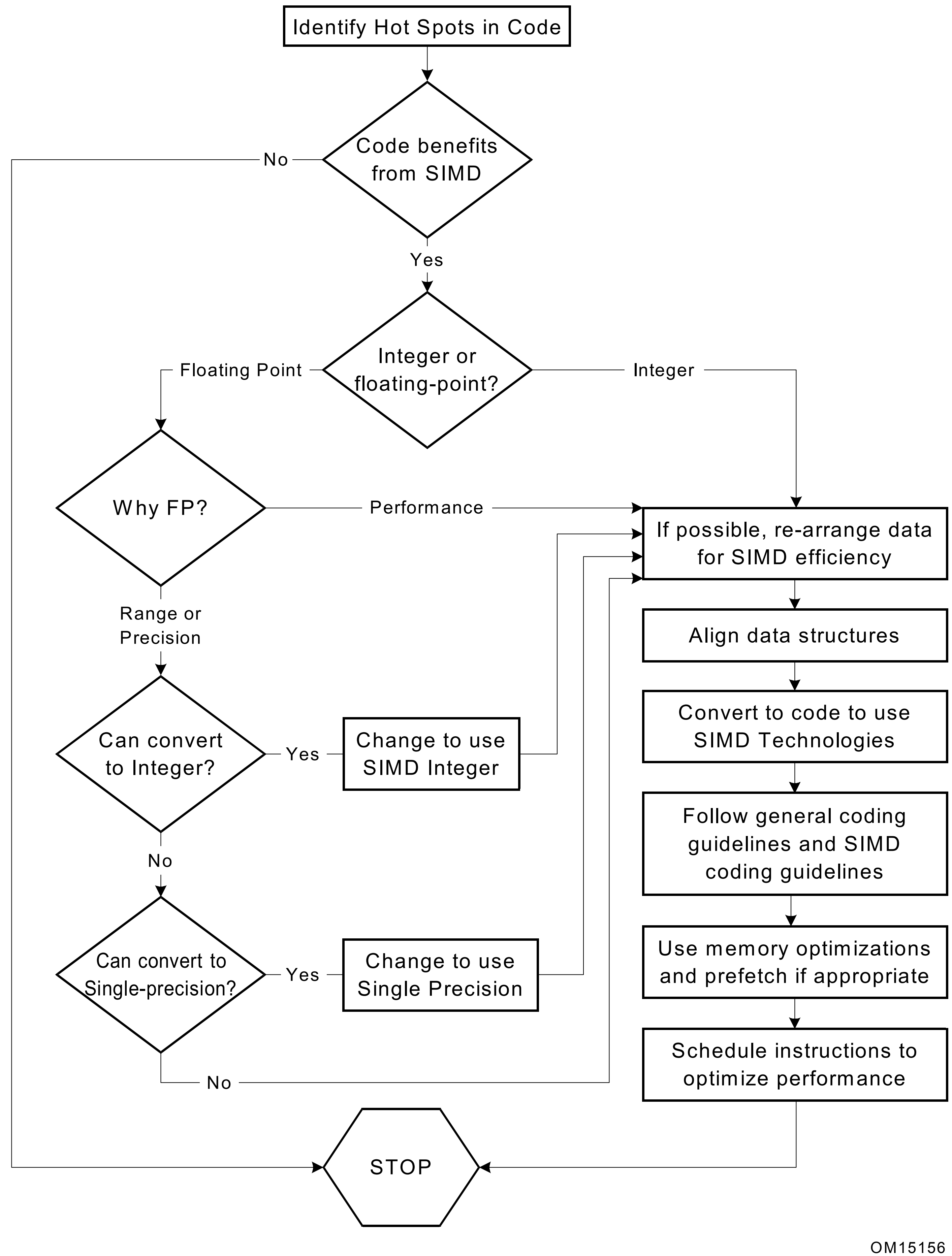
SIMD is found everywhere !

- In contemporary CPU architectures of all segments :
  - ✓ **High-performance general purpose processors**
    - ❖ Intel's MMX, SSE, SSE2, SSE3, SSE4
    - ❖ AMD's 3DNow!
    - ❖ SPARC's VIS and VIS2
    - ❖ Sun's MAJC
  - ✓ **Dedicated processors**
    - ❖ Cell Processor → Playstation
  - ✓ **Mobile processors**
    - ❖ ARM's NEON technology
    - ❖ MIPS MDMX (MaDMaX) and MIPS-3D
    - ❖ Exclusive FP or SIMD



SIMD usage chart

- Do you really need FP?
- What is the FP size? (do you really need FP of that size)
- If convertible to any smaller data size → **bingo** !
- Is it vectoriziable ?
- Align data in memory to enable efficient data transfer
- Program in SIMD …



SIMD usage overview

- **Disadvantages**
  - ✓ Not all the algorithms can benefit from data parallelism
  - ✓ Register files are costly area wise
  - ✓ There is an area overhead for the SIMD support of the ALUs (although very limited)
  - ✓ If the program is referencing data in non-contiguous manner, this might be a problem
    (in some cases it can be solved, but not always)
  - ✓ Can't be used automatically:
    - ❖ Vectorisation is possible only with some specific compilers that can
- **Advantages**
  - ✓ Computational speed
  - ✓ Libraries exist that implement important functionalities

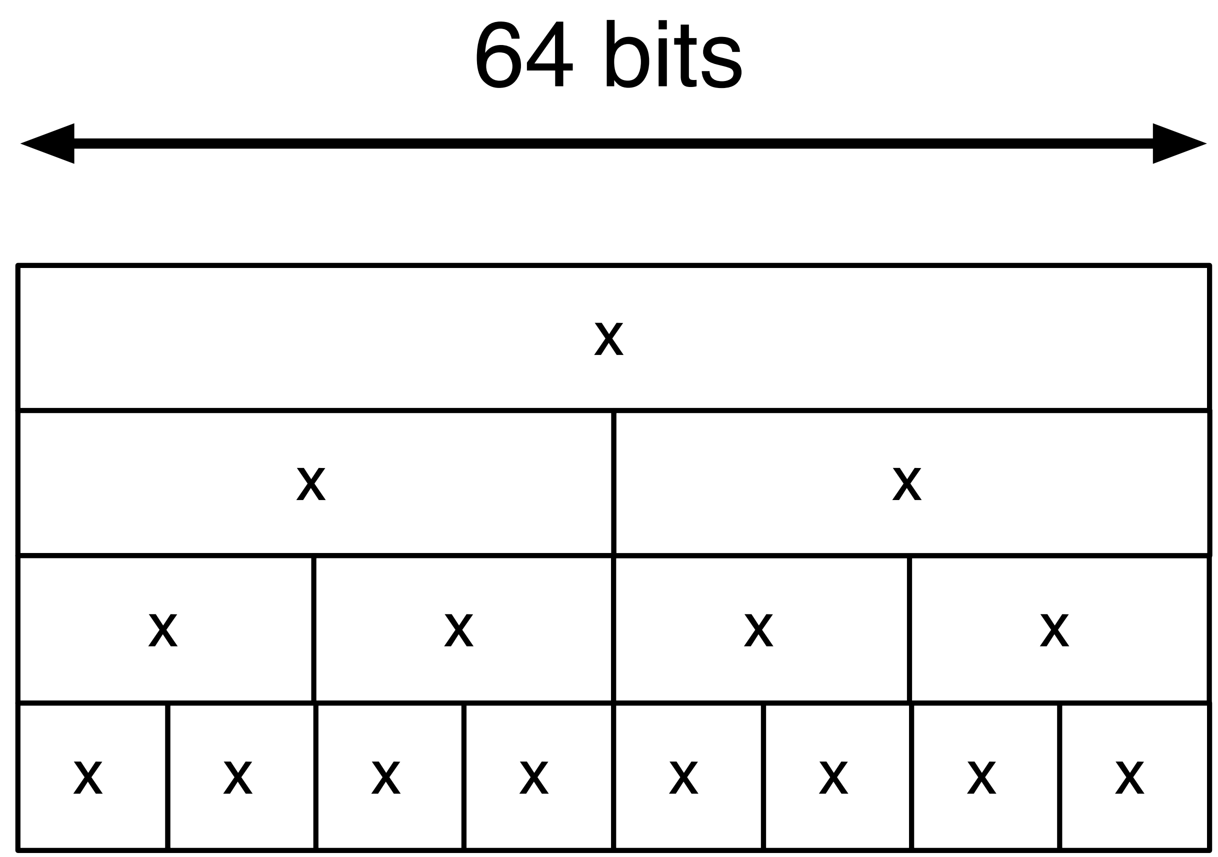
SIMD at Intel's x86

Overview

- Few generations of the extensions depending on the CPU architecture, having different code names :
  - ✓ **MMX** – in the beginnings
  - ✓ **SSE** – in the middle of the road
  - ✓ **AVX** – today
- Different extensions differ in :
  - ✓ **operand size** — increasing from 64 to 256 bits in ~10 years !
  - ✓ **associated instruction set** — as you can imagine increasing
- All extensions are **backward compatible**
  - ✓ CPU with AVX extension will run MMX, but it won't work the other way around

MMX — MultiMedia eXtension

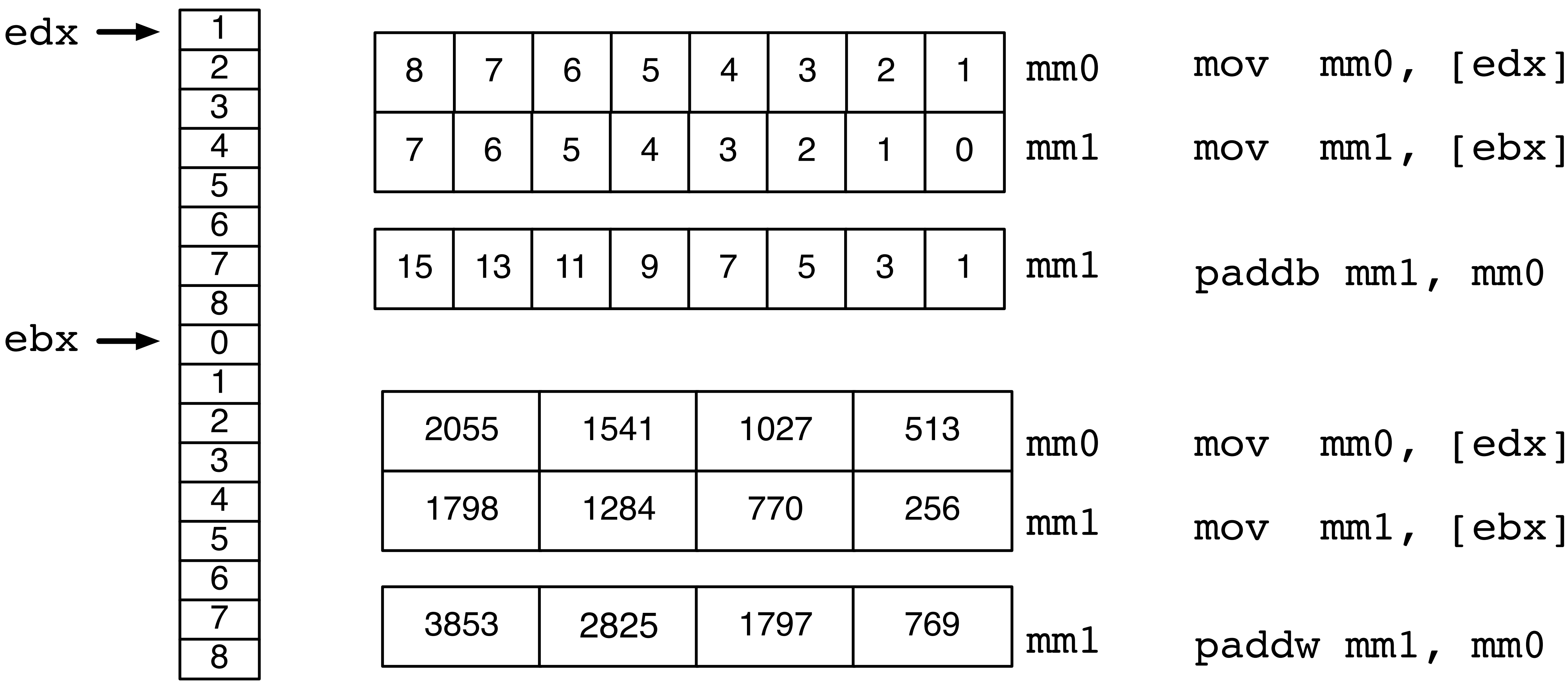
- AKA Multiple Math eXtension, Matrix Math eXtension
- First SIMD introduced in 1997 (Pentium CPUs)
- Idea
  - ✓ Reuses the FP register file through aliasing
  - ✓ FP registers are 80 bits → 64 bits will be used to store **packed data :**
    - ❖ single 64-bit integer,
    - ❖ two 32-bit integers,
    - ❖ four 16-bit integers,
    - ❖ eight 8-bit integers



  - ✓ Exclusive FP or SIMD (so no FP/SIMD concurrent operation)

Operation illustration

- Instructions are data type dependent, the memory is data type blind :



SSE — Streaming SIMD Extension

- Evolution of MMX introduced in 1999 (Pentium3)
- New 128 bit wide registers called : **xmm0-xmm7**
  (they are not any more aliased FP registers)
- Floating point instructions added to instruction set, so :
  - ✓ Data movements
    - ❖ **MOVAPS, MOVUPS,** MOVLPS, MOVHPS, MOVLHPS, MOVHLPS
  - ✓ Arithmetic
    - ❖ ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS
  - ✓ Compare
    - ❖ CMPSS, COMISS, UCOMISS, CMPPS
  - ✓ Data shuffle and unpacking
    - ❖ CVTSI2SS, CVTSS2SI, CVTTSS2SI
  - ✓ Bitwise logical operations
    - ❖ ANDPS, ORPS, XORPS, ANDNPS

SSE2, SSE3 & SSE4

- SSE2 Introduced in 2001 (Pentium4)
  - ✓ New math instructions for double precision (64-bit) floating point
  - ✓ SIMD operations on any data type : from 8-bit integer to 64-bit float entirely with the XMM vector-register file, without the need to use the legacy MMX or FPU registers
- SSE3 introduced in 2005 with Prescott variant of Pentium4
  - ✓ Adds some specific Digital Signal Processing & 3D instructions
  - ✓ More instructions that enable easy manipulation of the words inside the register
  - ✓ Better transfer for unaligned memory access
- SSE4 from 2006 in various flavors
  - ✓ … more instructions not necessarily multi-media

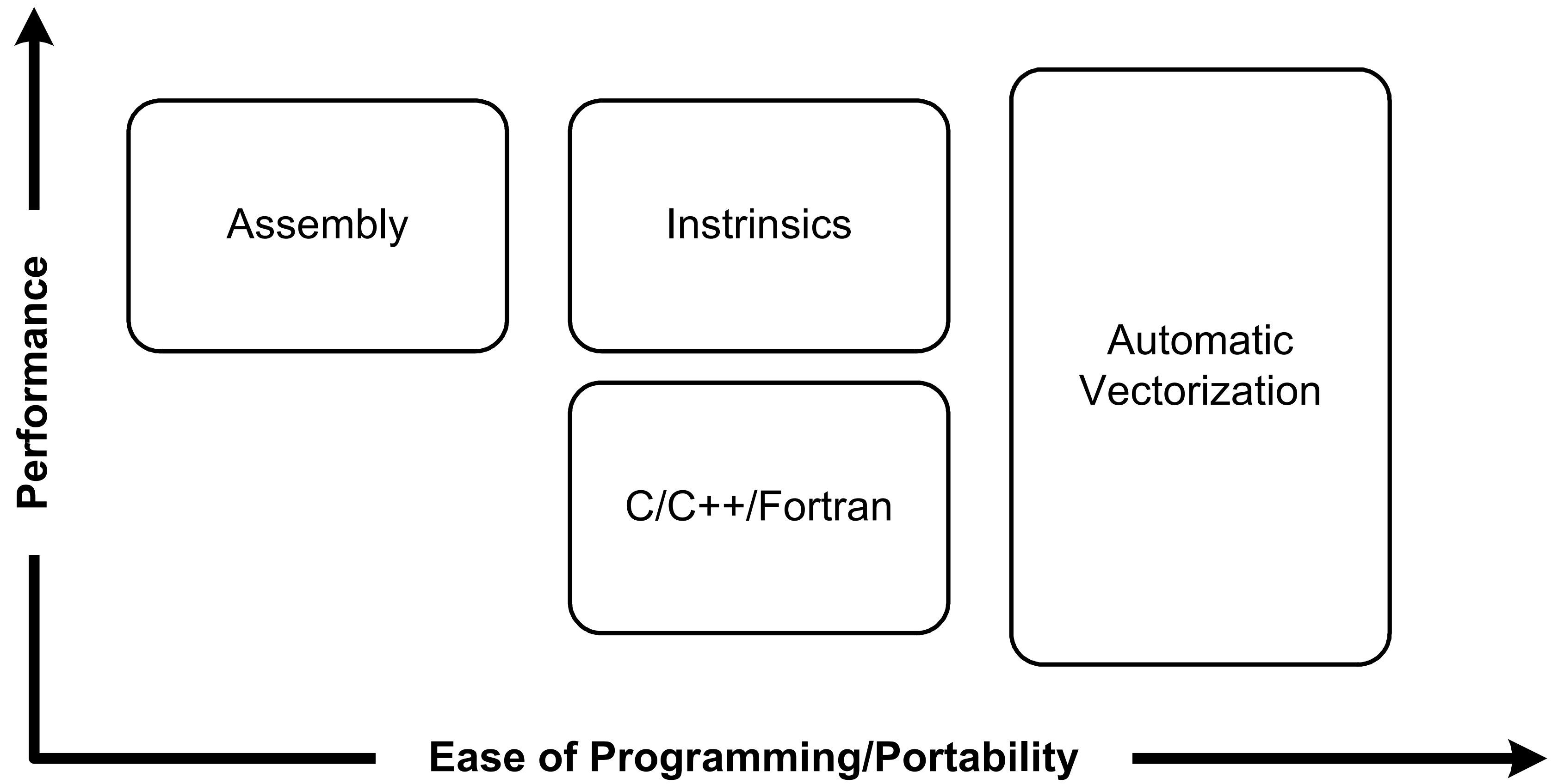
AVX — Advanced Vector Extensions

- Latest edition of SSE (changes the name) released in 2011
- Main features :
  - ✓ Data path increases from 128 bits to **256 bits**
  - ✓ Register file : 16x256 words named `ymm0` to `ymm15`
  - ✓ 2 operand instructions (1 source and 1 destination have to be the same) replaced with **3-operand instructions** :
    - ❖ `xmm0 ← xmm0 + xmm1` — erases the content of `xmm0`
    - ❖ `xmm2 ← xmm0 + xmm1` — both `xmm0` & `xmm1` remain untouched
- AVX requires support from the operating system, cannot be used with older operating systems like Windows XP or Windows Vista, even if the CPU supports AVX !

How to use SIMD on Intel CPUs ?

How to use SIMD in Intel CPUs ?

- Various options, but with very different performance/ease of use trade-off :



  - ✓ **Assembly**
  - ✓ **Intrinsics**
  - ✓ Intel C/C++ Classes
  - ✓ Libraries
    - ❖ Just need to get them and interface them with whatever code you want (since they are generic functions could be even further optimised, but best possible time-to-market)
  - ✓ **Automatic vectorization** — some compilers do, but difficult !

In-line assembly

- Using assembly language and assembler directly
- In the context of more complex application using C/C++ in-line assembly at any point in the C/C++ code
- Consider the example of a simple loop that can be SIMD encoded for much faster execution:

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++)
   {
   c[i] = a[i] + b[i];
   }
}
```

```
void add(float *a, float *b, float *c)
{
__asm
  {
   mov   eax, a
   mov   edx, b
   mov   ecx, c
   movaps xmm0, XMMWORD PTR [eax]
   addps   xmm0, XMMWORD PTR [edx]
   movaps XMMWORD PTR [ecx], xmm0
  }
}
```

Intrinsics

- Are functions that map directly the **assembly instruction** into **C-like functions**
- The performance is as good as assembly code, but is much easier to write
- Intrinsic code is compatible across different compilers that support these functions

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++)
   {
   c[i] = a[i] + b[i];
   }
}
```

```
#include <xmmintrin.h>

void add(float *a, float *b, float *c)
{
   __m128 t0, t1;
   t0 = _mm_load_ps(a);
   t1 = _mm_load_ps(b);
   t0 = _mm_add_ps(t0, t1);
   _mm_store_ps(c, t0);
}
```

Practically

- Instruction set changes with every new extensions: it is enriched with new functionalities (old instructions are kept for backward compatibility reasons)
- If in the past you missed a function, and you blew your brain to figure out the vector algorithm for a given problem, in the next generation it might appear a single instruction to use that will solve all your nightmares (will give an example)
- No magic you need to read the documentation:
  - ✓ Intel® SSE4 Programming Reference provides all the details
  - ✓ Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - ✓ and/or google for more …

Examples

Threshold image

- Loop over all the pixels in the image
- Compare the current pixel value with threshold
- Assign Min if less, assign Max if greater then

Original image

Threshold at 127

Pseudo-code

```
imagesrc = read_image(file);
start_time = get_time ();
for (index i in [0, all_pixels_in_the_image])
{
   curent_pixel_value = imageptr[i];
   if (curent_pixel_value < treshold)
        then new_pixel_value = 0;
     else
        new_pixel_value = 255;
   imagedst[i] = new_pixel_value;
}
end_time = get_time ();
elapsed_time = end_time - start_time;
print elapsed_time;
```

Algorithm using vector computation

```
ptrin = src;
ptrout = dest;
ii=(sizex)*(sizey)/16;             // Set the counter
_asm {
   mov       esi   , ptrin;        // datain ptr of the line in register
   mov       ecx   , ii;           // counter
   mov       edi   , ptrout;       // dataout pointer
   mov       eax   , mask;         // treshold value pointer
   movapd    xmm7  , [eax];        // 1st pipe mask
l1:
   movapd     xmm0 , [esi];       // load first line
   pcmpeqb    xmm0 , xmm7;        // compare
   movapd    [edi] , xmm0;       // Move result to memory destination
   add        edi , 16;          // New result pointer
   add        esi , 16;          // New source pointer
   sub        ecx , 1;
jnz      l1;
```

Example 2

Image filtering

- Image is a collection of pixels
- Gray level image : each pixel is coded using 8 bits values
  - ✓ 0 — is pure black
  - ✓ 255 — is pure white
  - ✓ whatever is in between are the shades of gray
- Data type : `unsigned char`
- **Filtering process** : for each pixel of the input image we assign a function of the it's neighboring pixels
- Depending on the function
  - ✓ linear filters (FIR) — polynomial function
  - ✓ non-linear filters — maximum, median, minimum etc.

123	75	128
32	65	44
255	1	22

Non-linear filter examples

123	75	128
32	65	44
255	1	22

1, 22, 32
44, 65, 75
123,128,255

1, 22, 32
44, 65, 75
123,128,255

Original image

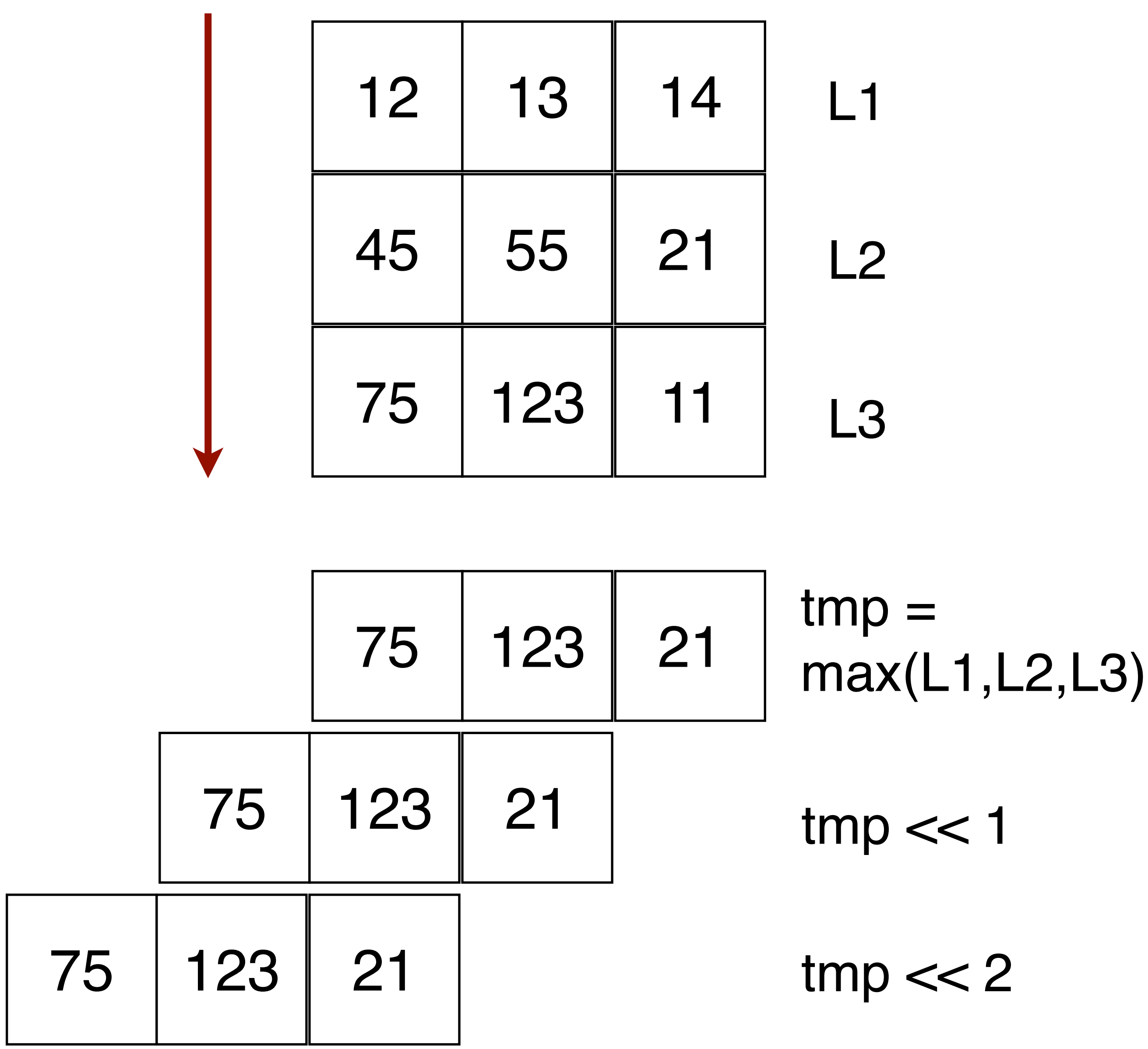
Median filter applied to the original image

Contour extraction using the difference between Max and Min filters

Max filter over the neighborhood

→ **SIMD Algorithm**

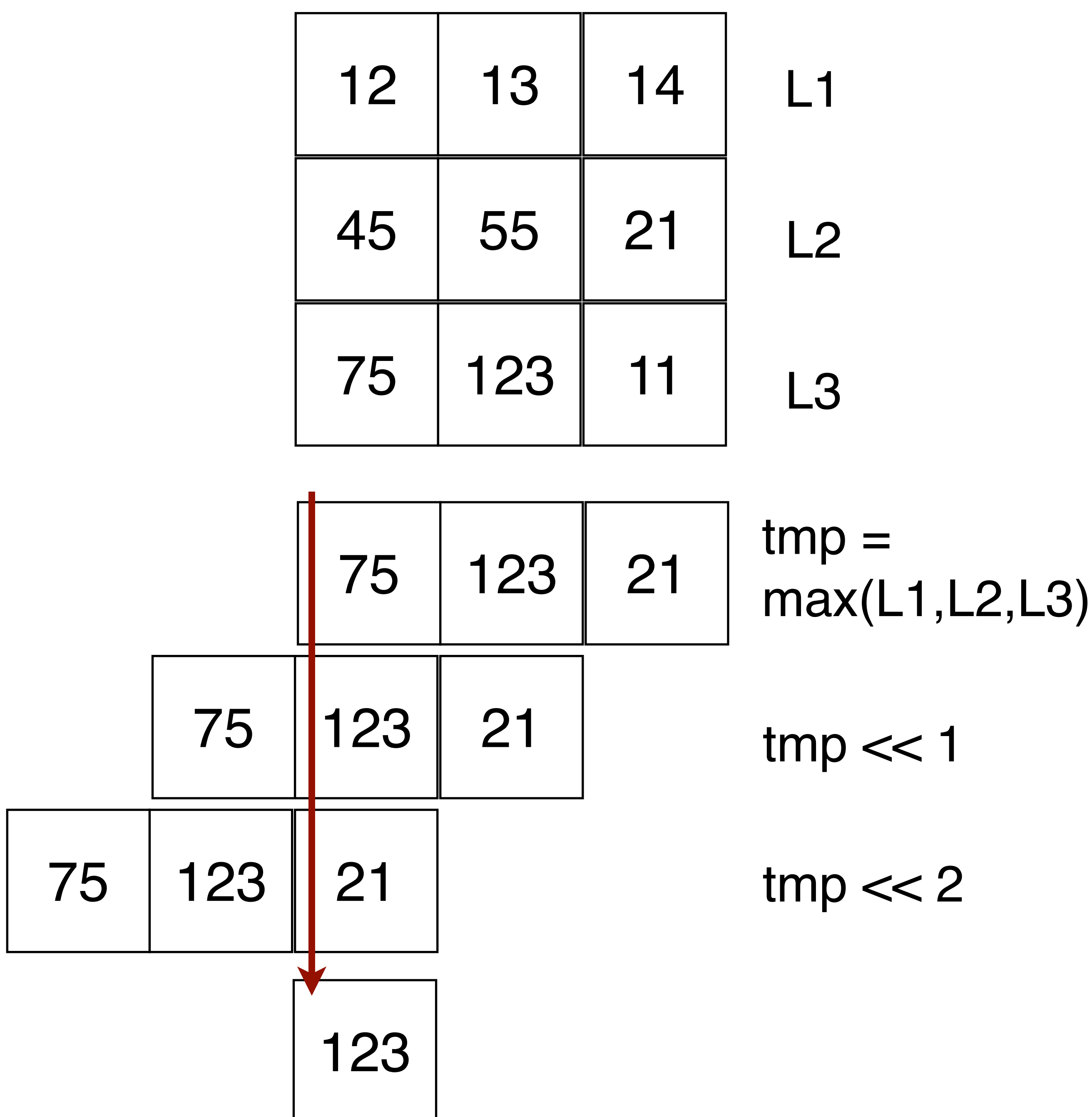
assuming 3x3 neighborhood size



- Take first three lines that correspond to one neighborhood and store them in the vector registers
- Compute max (or min) of different vectors — this is the computation of the max value on per column basis
- Store the value in some tmp register
- Shift the copy of the register on the left **once**
- Shift another copy of the register on the left **twice**

Max filter over the neighborhood

→ **SIMD Algorithm**
assuming 3x3 neighborhood size



- Compute max of the three different vectors — this is the computation of the max value of the neighborhood
- Store the value in some tmp register
- Shift the copy of the register on the left **once**
- Shift another copy of the register on the left **twice**

Assembly code assuming aligned mem

```
_asm {
   start:
      mov     esi , ptrin;       // datain ptr of the line
      mov     ecx , ii;          // counter
      mov     edi , ptrout;      // dataout pointer
l1:
  movdqu   xmm0 , [esi];        // l1st line
  movdqu   xmm1 , [esi+1024];   // 2nd line
  movdqu   xmm2 , [esi+2048];   // 3rd line
  pmaxub   xmm0 , xmm1;         // compare
  pmaxub   xmm0 , xmm2;
  movdqu   xmm6 , xmm0;         // copy
  movdqu   xmm7 , xmm0;
  psrldq   xmm6 , 1;            // shift
  psrldq   xmm7 , 2;
  pmaxub   xmm6 , xmm7;         // colon max
  pmaxub   xmm6 , xmm0;
  movdqu  [edi] , xmm6;         // move result
  add     edi , 14;             // new result pointer
  add     esi , 14;             // new src pointer
  sub     ecx , 1;
jnz   l1;                       // end
```

Max filter → SIMD computation

Control variables:
source and destination pointers
Loop counter

```
_asm {
   start:
      mov     esi , ptrin;       // datain ptr of the line
      mov     ecx , ii;          // counter
      mov     edi , ptrout;      // dataout pointer
l1:
  movdqu   xmm0 , [esi];         // l1st line
  movdqu   xmm1 , [esi+1024];    // 2nd line
  movdqu   xmm2 , [esi+2048];    // 3rd line
  pmaxub   xmm0 , xmm1;          // compare
  pmaxub   xmm0 , xmm2;
  movdqu   xmm6 , xmm0;          // copy
  movdqu   xmm7 , xmm0;
  psrldq   xmm6 , 1;             // shift
  psrldq   xmm7 , 2;
  pmaxub   xmm6 , xmm7;          // colon max
  pmaxub   xmm6 , xmm0;
  movdqu  [edi] , xmm6;          // move result
  add     edi , 14;              // new result pointer
  add     esi , 14;              // new src pointer
  sub     ecx , 1;
jnz   l1;                        // end
```

Max filter → SIMD computation

Loading 3 vectors of 16 into xmm0, xmm1 et xmm2

```
_asm {
   start:
      mov     esi , ptrin;       // datain ptr of the line
      mov     ecx , ii;          // counter
      mov     edi , ptrout;      // dataout pointer
l1:
  movdqu   xmm0 , [esi];        // l1st line
  movdqu   xmm1 , [esi+1024];   // 2nd line
  movdqu   xmm2 , [esi+2048];   // 3rd line
  pmaxub   xmm0 , xmm1;         // compare
  pmaxub   xmm0 , xmm2;
  movdqu   xmm6 , xmm0;         // copy
  movdqu   xmm7 , xmm0;
  psrldq   xmm6 , 1;            // shift
  psrldq   xmm7 , 2;
  pmaxub   xmm6 , xmm7;         // colon max
  pmaxub   xmm6 , xmm0;
  movdqu  [edi] , xmm6;         // move result
  add     edi , 14;             // new result pointer
  add     esi , 14;             // new src pointer
  sub     ecx , 1;
jnz   l1;                       // end
```

Max filter → SIMD computation

Max local of columns

Local copies to save the results
Shifts to align vectors

```
_asm {
   start:
      mov     esi , ptrin;       // datain ptr of the line
      mov     ecx , ii;          // counter
      mov     edi , ptrout;      // dataout pointer
l1:
  movdqu   xmm0 , [esi];        // l1st line
  movdqu   xmm1 , [esi+1024];   // 2nd line
  movdqu   xmm2 , [esi+2048];   // 3rd line
  pmaxub   xmm0 , xmm1;         // compare
  pmaxub   xmm0 , xmm2;
  movdqu   xmm6 , xmm0;         // copy
  movdqu   xmm7 , xmm0;
  psrldq   xmm6 , 1;            // shift
  psrldq   xmm7 , 2;
  pmaxub   xmm6 , xmm7;         // colon max
  pmaxub   xmm6 , xmm0;
  movdqu  [edi] , xmm6;         // move result
  add     edi , 14;             // new result pointer
  add     esi , 14;             // new src pointer
  sub     ecx , 1;
jnz   l1;                       // end
```

Max filter → SIMD computation

```
_asm {
   start:
      mov     esi , ptrin;       // datain ptr of the line
      mov     ecx , ii;          // counter
      mov     edi , ptrout;      // dataout pointer
l1:
  movdqu   xmm0 , [esi];         // l1st line
  movdqu   xmm1 , [esi+1024];    // 2nd line
  movdqu   xmm2 , [esi+2048];    // 3rd line
  pmaxub   xmm0 , xmm1;          // compare
  pmaxub   xmm0 , xmm2;
  movdqu   xmm6 , xmm0;          // copy
  movdqu   xmm7 , xmm0;
  psrldq   xmm6 , 1;             // shift
  psrldq   xmm7 , 2;
  pmaxub   xmm6 , xmm7;          // colon max
  pmaxub   xmm6 , xmm0;
  movdqu  [edi] , xmm6;          // move result
  add     edi , 14;              // new result pointer
  add     esi , 14;              // new src pointer
  sub     ecx , 1;
jnz   l1;                        // end
```

Max of the columns

Max filter → SIMD computation

```
_asm {
    start:
        mov     esi , ptrin;       // datain ptr of the line
        mov     ecx , ii;          // counter
        mov     edi , ptrout;      // dataout pointer
l1:
  movdqu   xmm0 , [esi];          // l1st line
  movdqu   xmm1 , [esi+1024];     // 2nd line
  movdqu   xmm2 , [esi+2048];     // 3rd line
  pmaxub   xmm0 , xmm1;           // compare
  pmaxub   xmm0 , xmm2;
  movdqu   xmm6 , xmm0;           // copy
  movdqu   xmm7 , xmm0;
  psrldq   xmm6 , 1;              // shift
  psrldq   xmm7 , 2;
  pmaxub   xmm6 , xmm7;           // colon max
  pmaxub   xmm6 , xmm0;
  movdqu  [edi] , xmm6;           // move result
  add     edi , 14;               // new result pointer
  add     esi , 14;               // new src pointer
  sub     ecx , 1;
jnz   l1;                         // end
```

Write results
Update pointers
Update loop counter
Jump