DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

COMPUTER ARCHITECTURES

SIMD processing

Gábor Horváth

BUTE Department of Networked Systems and Services

ghorvath@hit.bme.hu

Budapest,
05/05/2025

MŰEGYETEM 1782

FLYNN TAXONOMY

- Flynn's Taxonomy Based on the Relationship Between Instructions and Data
- **SISD** (Single Instruction, Single Data):
  - Execution of a single sequence of instructions on scalar data.
  - This is what we have studied so far.
- **SIMD** (Single Instruction, Multiple Data):
  - A single sequence of instructions operates on multiple data elements simultaneously.
  - Examples include vector processors, array processors, etc.
- **MIMD** (Multiple Instruction, Multiple Data):
  - Multiple sequences of instructions operate on multiple data elements independently.
  - Typical of multiprocessor systems.
- **MISD** (Multiple Instruction, Single Data):
  - Used in fault-tolerant systems.

Vector processors

VECTOR PROCESSORS

- Instead of using classical scalar data types and operations
  - Vector data types
  - Vector processing instructions
- Every modern supercomputer has vector processing capabilities

EXAMPLE

- C code:

```
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];
```

- Classical (scalar) solution:

```
R4←64
loop:
    D1 ← MEM[R1]
    D2 ← MEM[R2]
    D3 ← D1 + D2
    MEM[R3] ← D3
    R1 ← R1 + 8
    R2 ← R2 + 8
    R3 ← R3 + 8
    R4 ← R4 - 1
    JUMP loop IF R4!=0
```

Vector-based solution:

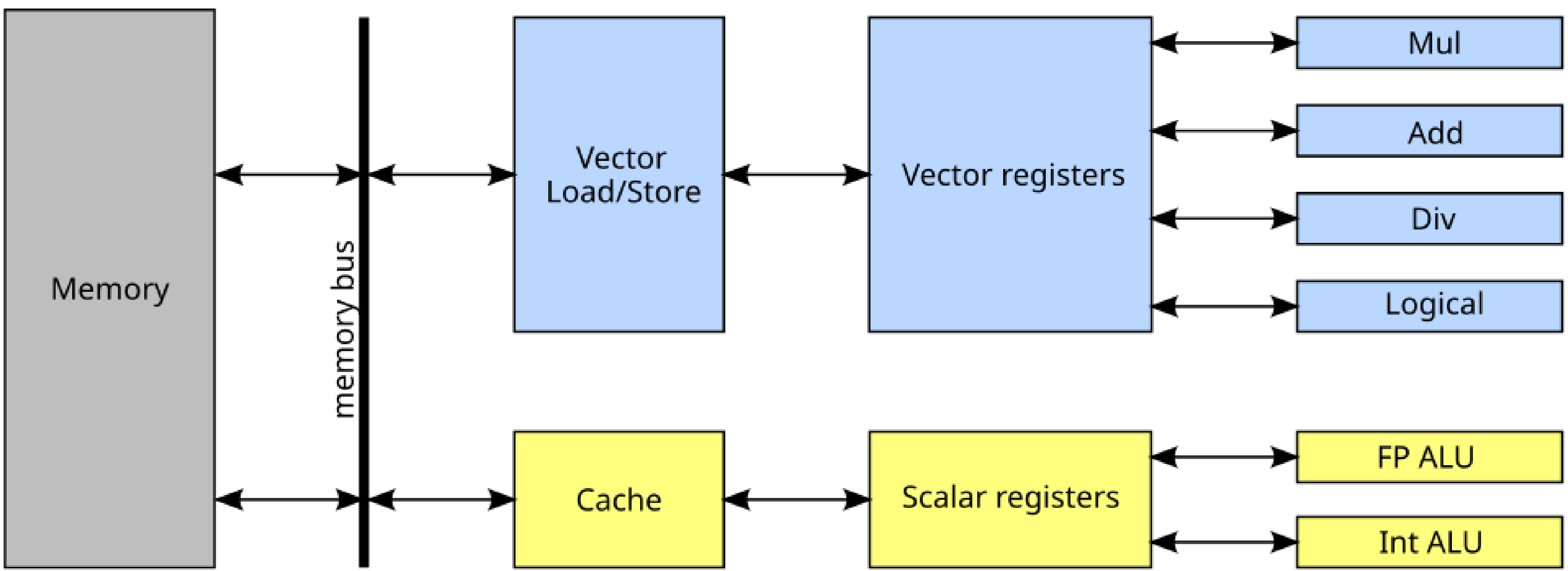
```
VLR ← 64
V1 ← MEM[R1]
V2 ← MEM[R2]
V3 ← V1 + V2
MEM[R3] ← V3
```

EXAMPLE

- Why is the vector-based solution better?
  - Shorter, more coincise code
  - **No loops are needed!**
    - What is wrong with loops?
      - In each cycle again and again the CPU has to
        - Fetch the instructions of the body
        - Decode them
        - Execute them
      - There is a control hazard in every iteration
      - Branch prediction is needed 64 times
  - **Vector instructions implicitly assume the independence of vector elements**
    - High performance can be achieved with very deep pipeline and/or multiple functional units

ARCHITECTURE OF VECTOR PROCESSORS

- Type of vector processors:
  - Register-register
  - Memory-memory
- We'll consider register-register vectorprocessors only



OPERATION OF VECTOR PROCESSORS

- Why do only scalar operands benefit from cache?
  - Vector operations are accelerated differently:
    → Through specialized memory handling.
- Why is a scalar memory read slow?
  - There are multiple clock cycles between issuing the address and the data arriving.
- Why are vector memory operations more efficient?
  - Vector loading involves a fixed (large) number of memory operations:
  - You issue the address of one element.
  - The data doesn't arrive in the next cycle yet, but you can already issue the address of the next element.
  - …and so on → a pipeline-like solution.
  - For this to work efficiently:
    Successive elements must be located in different memory banks.

INTERLEAVED MEMORY ADRESSING

Cycle	Bank					
1		15636				
2		Busy	15640			
3		Busy	Busy	15644		
4		Busy	Busy	Busy	15648	
5		Data[0]	Busy	Busy	Busy	15652
6	15656		Data[1]	Busy	Busy	Busy
7	Busy	15660		Data[2]	Busy	Busy
8	Busy	Busy	15664		Data[3]	Busy
9	Busy	Busy	Busy	15668		Data[4]
10	Data[5]	Busy	Busy	Busy	15672	
11		Data[6]	Busy	Busy	Busy	15676
12	15680		Data[7]	Busy	Busy	Busy
13	Busy	15684		Data[8]	Busy	Busy

FUNCTIONAL UNITS

- Accelerating vector operations:
- **By replicating functional units**
  - Vector elements are independent
  - e.g., with 4 functional units, 4 elements can be processed in parallel
- **By using a deep data pipeline**
  - Data pipeline? What's that?
  - Floating-point number representation: Number = (-1)^s × c × 2^q
  - Example: Floating-point addition (4 stages):
    - Check if either operand is zero
    - Align the two operands to the same exponent
    - Perform the addition
    - Normalize the result

DATA PIPELINE

- Example: Floating-Point Multiplication (5 Stages)
  - Check if either operand is zero
  - Add the exponents
  - Multiply the mantissas
  - Determine the sign bit of the result
  - Normalize the result
- As soon as the first stage is complete for one vector element, the processor immediately starts the first stage for the next element → pipelining

DATA PIPELINE

	1	2	3	4	5	6	7	8	9	10	11
V2[0] ← V0[0]+V1[0]	A0	A1	A2	A3							
V2[1] ← V0[1]+V1[1]		A0	A1	A2	A3						
V2[2] ← V0[2]+V1[2]			A0	A1	A2	A3					
V2[3] ← V0[3]+V1[3]				A0	A1	A2	A3				
V2[4] ← V0[4]+V1[4]					A0	A1	A2	A3			
V2[5] ← V0[5]+V1[5]						A0	A1	A2	A3		
V2[6] ← V0[6]+V1[6]							A0	A1	A2	A3	
V2[7] ← V0[7]+V1[7]								A0	A1	A2	A3

DATA PIPELINE

- With 2 pipelines:

	1	2	3	4	5	6	7	8	9	10	11
V2[0] ← V0[0]+V1[0]	A0	A1	A2	A3							
V2[1] ← V0[1]+V1[1]	A0	A1	A2	A3							
V2[2] ← V0[2]+V1[2]		A0	A1	A2	A3						
V2[3] ← V0[3]+V1[3]		A0	A1	A2	A3						
V2[4] ← V0[4]+V1[4]			A0	A1	A2	A3					
V2[5] ← V0[5]+V1[5]			A0	A1	A2	A3					
V2[6] ← V0[6]+V1[6]				A0	A1	A2	A3				
V2[7] ← V0[7]+V1[7]				A0	A1	A2	A3				

DATA PIPELINE

- Very Important:
  - This is a type of pipeline where
    - **There is no data hazard!**

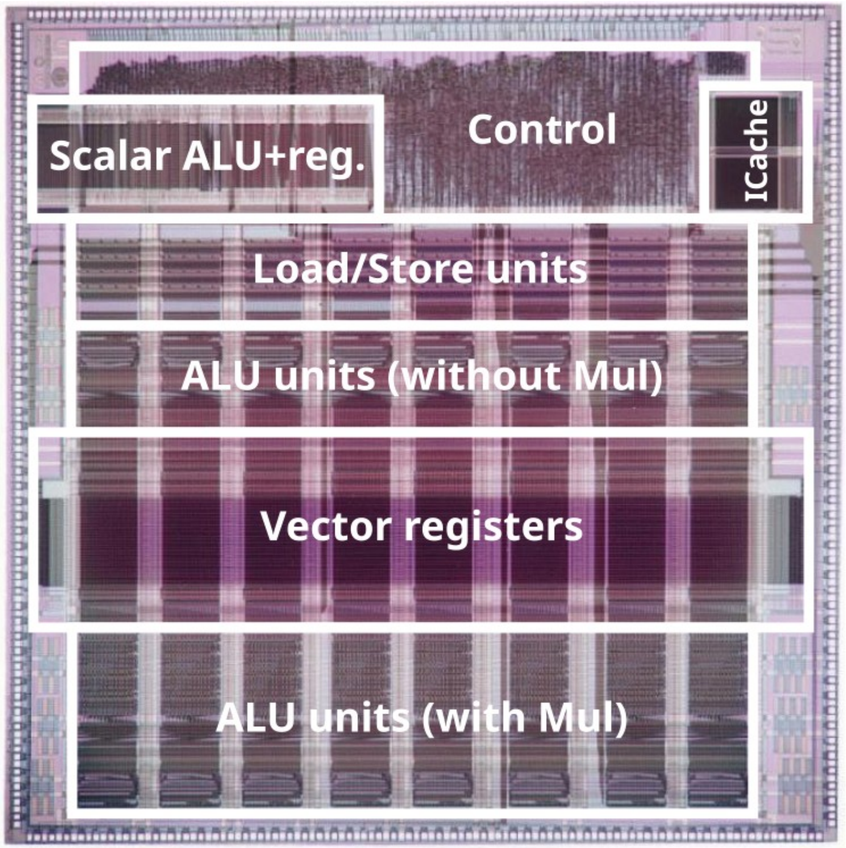
      → No need for waiting or hazard detection logic

      → This allows for arbitrarily deep pipelines

      → And arbitrarily wide pipelines
  - The only limiting factor:
    - How many sub-stages we can divide an arithmetic operation into

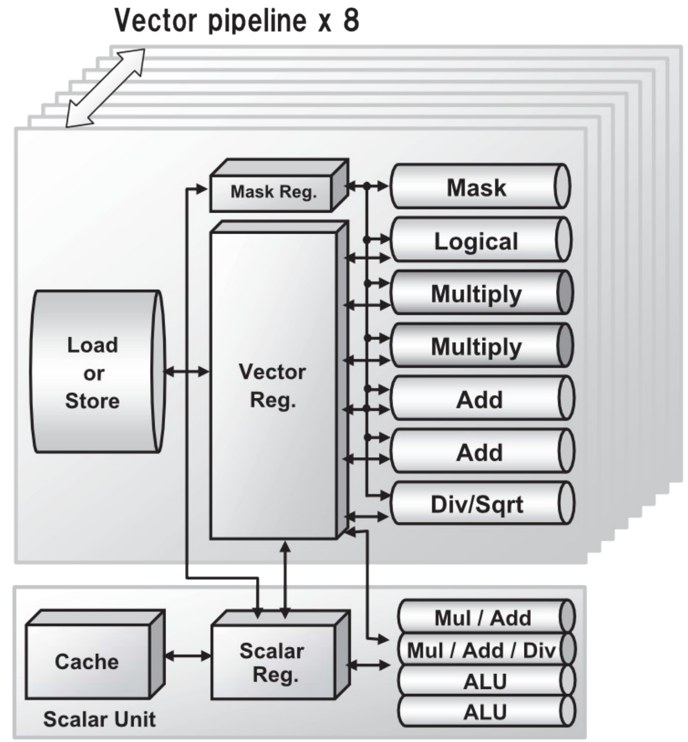
EXAMPLE 1.

- Berkely T0 (Torrent-0, 1995)
- Length-32 vector registers
- 1 lane: responsible for 4 elements (8 lines on the figure)
- 2 ALUs in each lane
- ...only one of them can multiply



EXAMPLE 2.

- NEC SX-9 Supercomputer Processor
- The fastest supercomputer of 2008
- Used by:
  German Meteorological Center (2 units)
- In 2011: 976 vector processors
  + 31 TB of memory
- Per processor:
  - Handles vectors of length 256
  - Contains 8 pipelined functional units
- Vector ALU runs at 3.2 GHz
- Scalar unit: 4-way out-of-order
  execution at 1.6 GHz



Typical Solutions in Vector Processors

STRIP-MINING

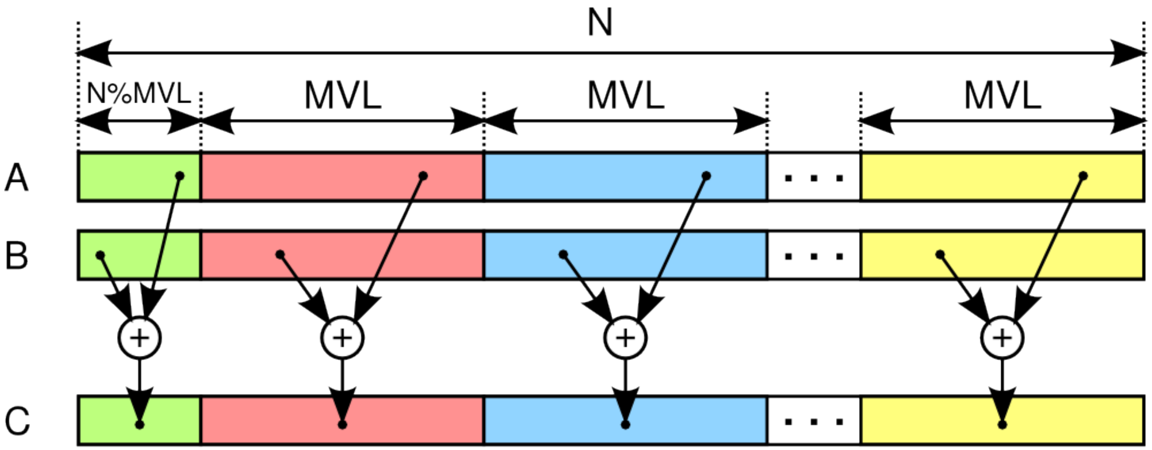
- Hardware-Supported Vector Length is Fixed: MVL (Maximum Vector Length)
- We rarely need vectors exactly this size.
  - If we compute with a smaller vector:
    - Set the VLR (Vector Length Register)
    - Smaller VLR → shorter execution time
  - If we compute with a larger vector:
    - The vector is split into MVL-sized chunks
    - The operation is executed on each chunk

  → This is called **Strip-mining**

STRIP-MINING EXAMPLE

- C code:

```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
```



Vectorized:

```
   VLR ← R0 % MVL
loop:
   V1 ← MEM[R1]
   V2 ← MEM[R2]
   V3 ← V1 + V2
   MEM[R3] ← V3
   R4 ← VLR * 8
   R1 ← R1 + R4
   R2 ← R2 + R4
   R3 ← R3 + R4
   R0 ← R0 - VLR
   VLR ← MVL
   JUMP loop IF R0!=0
```

VECTOR MASKING

- There is a special mask register
- ALU skips vector elements where mask=0
- C code:

```
for (i=0; i<N; i++)
    if (B[i]>0)
        C[i] = A[i] / B[i];
```

Vectorized:

```
V1 ← MEM[R1]
MASK ← V1>0
V0 ← MEM[R0]
V2 ← V0 / V1
MEM[R2] ← V2
```

- Two possible implementations:
  - **Naive**: Vector ALU applies operation on all elements, but it does not store masked out elements
  - **Efficient**: The Load/Store unit and the ALU skips masked out elements → faster execution

VECTOR CHAINING

- RAW dependencies exist in vector processors, too:
- ```
  V1 ← MEM[R1]
  V3 ← V1 + V2
  V5 ← V3 * V4
  ```

- A solution in vector processors similar to forwarding:
  - **Vector chaining**
  - Dependent instruction does not have to wait for the one providing the operand
  - As soon as the first element of V1 is available, we can start working on the first element of V3
  - As soon as the first element of V3 is available, we can start working on the first element of V5
  ```

EXAMPLE FOR VECTOR CHAINING

Instruction	1	2	3	4	5	6	7	8	9
V1[0]←MEM[R1+0]	T0	T1							
V1[1]←MEM[R1+8]	T0	T1							
V1[2]←MEM[R1+16]		T0	T1						
V1[3]←MEM[R1+24]		T0	T1						
V1[4]←MEM[R1+32]			T0	T1					
V1[5]←MEM[R1+40]			T0	T1					
V3[0]←V1[0]+V2[0]			A0	A1	A2				
V3[1]←V1[1]+V2[1]			A0	A1	A2				
V1[6]←MEM[R1+48]				T0	T1				
V1[7]←MEM[R1+56]				T0	T1				
V3[2]←V1[2]+V2[2]				A0	A1	A2			
V3[3]←V1[3]+V2[3]				A0	A1	A2			
...									

EXAMPLE FOR VECTOR CHAINING

Instruction	1	2	3	4	5	6	7	8	9
V1[8]←MEM[R1+64]					T0	T1			
V1[9]←MEM[R1+72]					T0	T1			
V3[4]←V1[4] + V2[4]					A0	A1	A2		
V3[5]←V1[5] + V2[5]					A0	A1	A2		
V1[10]←MEM[R1+80]						T0	T1		
V1[11]←MEM[R1+88]						T0	T1		
V3[6]←V1[6] + V2[6]						A0	A1	A2	
V3[7]←V1[7] + V2[7]						A0	A1	A2	
V5[0]←V3[0] * V4[0]						M0	M1	M2	M3
V5[1]←V3[1] * V4[1]						M0	M1	M2	M3
...									

SIMD instruction set extensions

SIMD INSTRUCTION SET EXTENSIONS

- Vector instructions are useful even for consumer use
  - Useful for image processing
  - In 3D graphics applications and games
  - Even simple scientific computations can be well-vectorized
- Many processors support vector operations
- But that doesn’t make them true vector processors!
  - Why not?
    - Vector size is very small (at best: 256 bits)
    - No Vector Length Register (VLR)
    - No mask registers
    - No vector chaining
    - No data pipeline
      - The number of functional units = vector size

SIMD INSTRUCTION SET EXTENSIONS

- Instruction types:
  - Vector-vector operations
    - Inter-vector: between 2 vectors. Result: vector
      - For example: adding two vectors
    - Intra-vector: on the elements of a vector. Result: scalar
      - For example: summing the elements of a vector
    - Reordering elements of a vector (shuffling)
  - Scalar-vector operations:
    - For example: multiplying each vector element by a scalar
  - Vector load/store operations:
    - Memory ↔ vector register data transfer

SIMD
INSTRUCTION SET EXTENSIONS

Name	ISA	Num. vec. regs.	Length	Type of elements
MMX	x86	8	64 bit	Int: 8x8, 4x16, 2x32 bit
3DNow	x86	8	64 bit	Float: 2x32 bit
SSE	x86/x64	8	128 bit	Float: 4x32 bit
SSE2-4	x86/x64	8/16	128 bit	Int: 16x8, 8x16, 4x32 bit. Float: 4x32, 2x64 bit
AVX	x86/x64	16	256 bit	Float: 8x32, 4x64 bit
AltiVec	Power	32	128 bit	Int: 16x8, 8x16, 4x32 bit Float: 4x32 bit
NEON	ARM	32/16	64/128 bit	Int: 8x8, 4x16, 2x32 bit Float: 2x32 bit

SIMD OPERATIONS IN HIGH LEVEL LANGUAGES

- SIMD operations can be used in high level languages, too!

  → „intrinsic" instructions

  - Platform dependent
- Usage:
  - Include the appropriate C header file
    - Introduces vector data types (`__m128`, `float32x4_t`)
    - Introduces vector operations

EXAMPLE FOR SIMD IN HIGH LEVEL LANGUAGE

- Increasing image saturation, without SIMD:

```
void saturate () {

    float r, g, b, p, val;

    for (int i=0; i<height*width; i++) {
        r = *srcR;
        g = *srcG;
        b = *srcB;

        p = sqrt (r*r + g*g + b*b);
        val = p + (r - p) * 1.5f;
        *dstR = val>255.0 ? 255.0 : val<0 ? 0 : val;
        val = p + (g - p) * 1.5f;
        *dstG = val>255.0 ? 255.0 : val<0 ? 0 : val;
        val = p + (b - p) * 1.5f;
        *dstB = val>255.0 ? 255.0 : val<0 ? 0 : val;

        srcR++; srcG++; srcB++;
        dstR++; dstG++; dstB++;
    }
}
```

WITH SSE2 INSTRUCTIONS

```
#include <xmmintrin.h>
void saturateSSE2 () {

    float p, val;
    __m128 r0, r1, r2, r3, r4;
    const __m128 r5 = {1.5f, 1.5f, 1.5f, 1.5f};
    const __m128 r6 = {0.0f, 0.0f, 0.0f, 0.0f};
    const __m128 r7 = {255f, 255f, 255f, 255f};

    for (int i=0; i<height*width; i+=4) {

        r1 = _mm_load_ps (srcR);
        r0 = r1;
        r0 = _mm_mul_ps (r0, r1);

        r2 = _mm_load_ps (srcG);
        r4 = r2;
        r4 = _mm_mul_ps (r4, r2);
        r0 = _mm_add_ps (r0, r4);

        r3 = _mm_load_ps (srcB);
        r4 = r3;
        r4 = _mm_mul_ps (r4, r3);
        r0 = _mm_add_ps (r0, r4);

        r0 = _mm_sqrt_ps (r0);
```

```
        r1 = _mm_sub_ps (r1, r0);
        r1 = _mm_mul_ps (r1, r5);
        r1 = _mm_add_ps (r1, r0);
        r1 = _mm_min_ps (r1, r7);
        r1 = _mm_max_ps (r1, r6);

        _mm_store_ps (dstR, r1);

        r2 = _mm_sub_ps (r2, r0);
        r2 = _mm_mul_ps (r2, r5);
        r2 = _mm_add_ps (r2, r0);
        r2 = _mm_min_ps (r2, r7);
        r2 = _mm_max_ps (r2, r6);

        _mm_store_ps (dstG, r2);

        r3 = _mm_sub_ps (r3, r0);
        r3 = _mm_mul_ps (r3, r5);
        r3 = _mm_add_ps (r3, r0);
        r3 = _mm_min_ps (r3, r7);
        r3 = _mm_max_ps (r3, r6);

        _mm_store_ps (dstB, r3);

        srcR+=4; srcG+=4; srcB+=4;
        dstR+=4; dstG+=4; dstB+=4;
    }
}
```

WITH NEON INSTRUCTIONS

```
#include <arm_neon.h>
void saturateNEON () {

    float p, val;
    float32x4_t r0, r1, r2, r3, r4;
    const float32x4_t r5 = vdupq_n_f32 (1.5f);
    const float32x4_t r6 = vdupq_n_f32 (0.0f);
    const float32x4_t r7 = vdupq_n_f32 (255.0f);

    for (i=0; i<height*width; i+=4) {

        r1 = vld1q_f32 (srcR);
        r0 = vmulq_f32 (r1, r1);

        r2 = vld1q_f32 (srcG);
        r4 = vmulq_f32 (r2, r2);
        r0 = vaddq_f32 (r0, r4);

        r3 = vld1q_f32 (srcB);
        r4 = vmulq_f32 (r3, r3);
        r0 = vaddq_f32 (r0, r4);

        r0 = vrecpeq_f32 (vrsqrteq_f32 (r0));

        r1 = vsubq_f32 (r1, r0);
        r1 = vmulq_f32 (r1, r5);
        r1 = vaddq_f32 (r1, r0);
        r1 = vminq_f32 (r1, r7);
        r1 = vmaxq_f32 (r1, r6);

        vst1q_f32 (dstR, r1);

        r2 = vsubq_f32 (r2, r0);
        r2 = vmulq_f32 (r2, r5);
        r2 = vaddq_f32 (r2, r0);
        r2 = vminq_f32 (r2, r7);
        r2 = vmaxq_f32 (r2, r6);

        vst1q_f32 (dstG, r2);

        r3 = vsubq_f32 (r3, r0);
        r3 = vmulq_f32 (r3, r5);
        r3 = vaddq_f32 (r3, r0);
        r3 = vminq_f32 (r3, r7);
        r3 = vmaxq_f32 (r3, r6);

        vst1q_f32 (dstB, r3);

        srcR+=4; srcG+=4; srcB+=4;
        dstR+=4; dstG+=4; dstB+=4;
    }
}
```

PERFORMANCE COMPARISON

- Execution times:

Intel Core i7-2600	Without SIMD	15,166 ms
	SSE2	3,829 ms
	AVX	3,698 ms
Intel Pentium 4	Without SIMD	139,758 ms
	SSE2	36,355 ms
ARM Cortex A9	Without SIMD	155,012 ms
	NEON	44,026 ms

DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

MŰEGYETEM 1782