INFOMOV 2019 FINAL EXAM - EDUC-THEATRON - 17:00 - 19:00

- The 'External Memory Model' assumes that the cost of running an algorithm is proportional to the number of memory blocks that need to be read into the cache during the execution of the algorithm. Describe one scenario in which this assumption is valid, and one scenario in which this assumption is invalid, or at least mostly invalid. Valid scenarios are plenty. Invalid scenarios: anything that loads some data and operates on it endlessly. The Fibonacci example that several brought up is excellent. Not acceptable: just making things more expensive (sin/cos/sqrt on a stream of data). In that case things are still proportional to the number of accessed blocks.
- 2. Consider the following data structure:

```
struct Nuke
{
    float x, y, z; // position
    bool homing; // flag
    float vx, vy, vz; // missile velocity
    bool exploded; // flag
};
```

Basic idea: you want to minimize memory cost, i.e. the # of touched cache lines. For random access this means aligning the struct to cache line boundaries (make it 2^N in size). For sequential access this means making it as small as possible.

Layouts for SIMD processing also yielded full points for sequential access.

Rewrite this structure twice: once for efficient <u>random access</u> of a large array of Nukes, and once for efficient <u>sequential access</u> of a large number of Nukes in a single, continuous array. Motivate your layouts. Some pitfalls for Q2: a float is automatically aligned to 4 bytes. A cacheline is not 128bit. A bool is not 1 bit.

3. Some CPU architecture questions, based on "Modern Microprocessors - a 90 minute guide":

In 30 words or less,

- a) explain what a *pipeline latch* is.
- b) explain what a *bypass* is.
- c) explain what *speculative execution* is.
- d) explain what *predication* is.

Skipped by most of you, but easily extracted from the document. Quite interesting actually. Still a recommended read. 😨

- 4. The Core i7-8700K processor uses 2x6MB of 16-way set-associative L3 cache. AMD's K10 uses 6MB of 48-way set-associative cache.
 - a) Under what circumstances is 48-way better than 16-way, and when and/or how is 16-way better?
 - b) Both processors use much lower set associativety for L1 and L2. Why do you think this is?

The whole class is cache expert, but for completeness:

48-way is closer to fully-associative and reduces collisions better than 16-way. It is also more costly in terms of die space, so that goes at the expense of other things. Also, more complex schemes tend to be slower, which applies here: Intel has the faster cache. Also: beyond 16-wide we get diminishing returns, and 48-way (even 64-way was attempted!) is over the top.

5. The following code uses fixed point arithmetic:

```
uint ScaleColor( const uint c, const uint x )
{
    uint redblue = c & 0x00ff00ff, green = c & 0x0000ff00;
    redblue = (redblue * x) & 0xff00ff00;
    green = (green * x) & 0x00ff0000;
    return (redblue + green) >> 8;
}
```

Most of you knew that blue bleeds into green if we don't take measures.

For 'b' the answer is 256, which I didn't expect myself (I thought it was simply not possible and used 255 as the best alternative).

This made the 'trick question' rather easy.

- a) Explain why the three color components are not multiplied by x using a single multiplication.
- b) Which value of x should we use to get the unscaled color, i.e. 100% of input value c?
- 6. Consider the following VTune profiling result for the 'rotozooming' example:

<pre>31 r[1].x = p[1].x * cosf(a) * p[1].y * sinf(a);</pre>				0x14	52	and eax, Oxff00	243,200,000
<pre>32 r[1].y = p[1].x * sinf(a) - p[1].y * cosf(a);</pre>				0x14	54	add edx, eax	70,400,000
33 }				0x14	53	mov eax, ebx	166,400,000
34 vec2 dx = (r[1] - r[0]) * ((sinf(a * 2.0f) + 1.1f) / SCRWIDTS);			_	0x14	53	imul eax, r8d	9,600,000
<pre>35 vec2 dy = (r[2] - r[0]) * ((sinf(a * 2.0f) + 1.1f) / SCRHEIGHT);</pre>			_	0x14	51	imul r10d, edi	201,600,000
36 for(int y = 0; y < SCRMEIGHT; y++)	3,200,000	3,200,000	1.000	0x14	51	shr eax, Cx8	32,000,000
37 (_	0x14	53	and eax, Oxff00ff	208,000,000
<pre>38 float x1 = dy.x * y, y1 = dy.y * y;</pre>				0x14	51	shr r10d, 0x8	28,800,000
39 for(int u = 0; u < SCRWIDTH; u++, x1 += dx.x, y1 += dx.y)	352,000,000	214,400,000	1.642	0x14	54	add edx, eax	198.400.000
40 (0x14	51	and r10d, 0xff00	105,600,000
41 float x = (x1 + 100) * 8192, y = (y1 + 100) * 8192;	476,800,000	368,000,000	1.296	0x14	52	mov eax, ebx	153,600,000
42 float fx = x - floor(x), fy = y - floor(y);	3,507,200,000	2,550,400,000	1.375	0x14	54	add edx, r10d	28.800.000
43 int ix = (int)x & 8191, iy = (int)y & 8191;	496,000,000	268,800,000	1.845	0x14	52	imul eax, rlld	227,200,000
44 float w1 = (1 - fx) * (1 - fy);	150,400,000	86,400,000	1.741	0x14	51	imul ebx, edi	32,000,000
45 float w2 = fx * (1 - fy);				0x14	51	shr eax, 0x8	140,800,000
46 float w3 = (1 - fx) * fy;				0x14	52	and eax, Oxff00ff	38.400.000
47 float wi = fx + fy;	499,200,000	405,400,000	1.228	0x14	51	shr ebx, 0x8	262,400,000
48 Pixel* hase = imageTest.GetBuffer();	22,400,000	19,200,000	1.167	0x14	54	add edx, eax	80,000,000
49 int offset = ix + iy * 8192;	908,800,000	710,400,000	1.279	0x14	51	and ebx, Oxff00ff	227.200.000
50 Fixel pl = ScaleColor(base(offset), (int)(wl * 255.9f));	652,800,000	867,200,000	0.753	0x14	54	add edx, ebx	32,000,000
51 Pixel p2 = ScaleColor(base(((offset + 1) & 67108863)), (int)(w2 *	55 43,603,200,000	39,673,600,000	1.099	0x14	54	mow dword ptr [r15], edx	240,000,000
52 Pixel p3 = ScaleColor(base(((offset + 1) & 67108863)), (int)(w3 *	55 1,568,000,000	1,225,600,000	1.279	0x14	54	add r15, Cx4	787,200,000
53 Fixel p4 = ScaleColor(base[((offset + 1) & 67108863)], (int)(w4 *	55 3,244,800,000	2,659,200,000	1.220	0x14	54	sub rbp, 0x1	
	1,920,000,000		1.167	0x14	54	ing 0x1400022e4 <block 75<="" td=""><td></td></block>	
55)				0x14		Block 14:	
56)				0x14	36	moveps mmn12, mmnword ptr	

Not accepted: the code fills up a latency of an expensive line before it and therefore takes 0 cycles. That may be true, but then the profiler could still sample in the cycles that make up the latency, which is not happening. Also, the lines would not be taking 0 cycles. Instead, the latency would appear smaller.

Also tricky. I *thought* 45 and 46 took little time and got overlooked by the random sampling of the profiler. This answer is still valid, as this could be the case based on the nfo in the image.

It is however also possible that the functionality is implemented as part of other asm lines, which point back to lines that are not 45 and 46.

The most plausible reason is that the compiler optimizes the lines away: the multiplication results are probably already in registers and are simply re-used.

- a) The first column to the right of the source code (in the red box) indicates the number of clock cycles spent on each line, according to the profiler. Apparently, line 45 and 46 did not take any cycles at all, but that seems unlikely. Explain what is going on.
- b) Some lines, such as line 50, are represented by a continuous block of assembler instructions.
 Other lines, such as line 54 (highlighted in blue), are compiled to scattered assembler instructions.
 Why is this?
- 7. "A just-in-time (JIT) compiler is a form of self-modifying code." Do you feel this statement is correct? Why or why not? (your answer should demonstrate that you understand the concept of self-modifying code). Not a great question. It assumes that you know what 'JIT' is, and if you do, it requires minimal understanding of the concept of SMC. Free points for most of you.
- 8. Sometimes a program becomes significantly *slower* when modified from single-threaded to multithreaded execution, due to *false sharing*. Write down a simple case where this happens. Heavy pseudo code is allowed (within reason), as long as your intention is clear.

(Almost) everyone was able to come up with a simple example, often based on P3 experience.