/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2016 - Lecture 14: “Grand Recap”

Welcome!

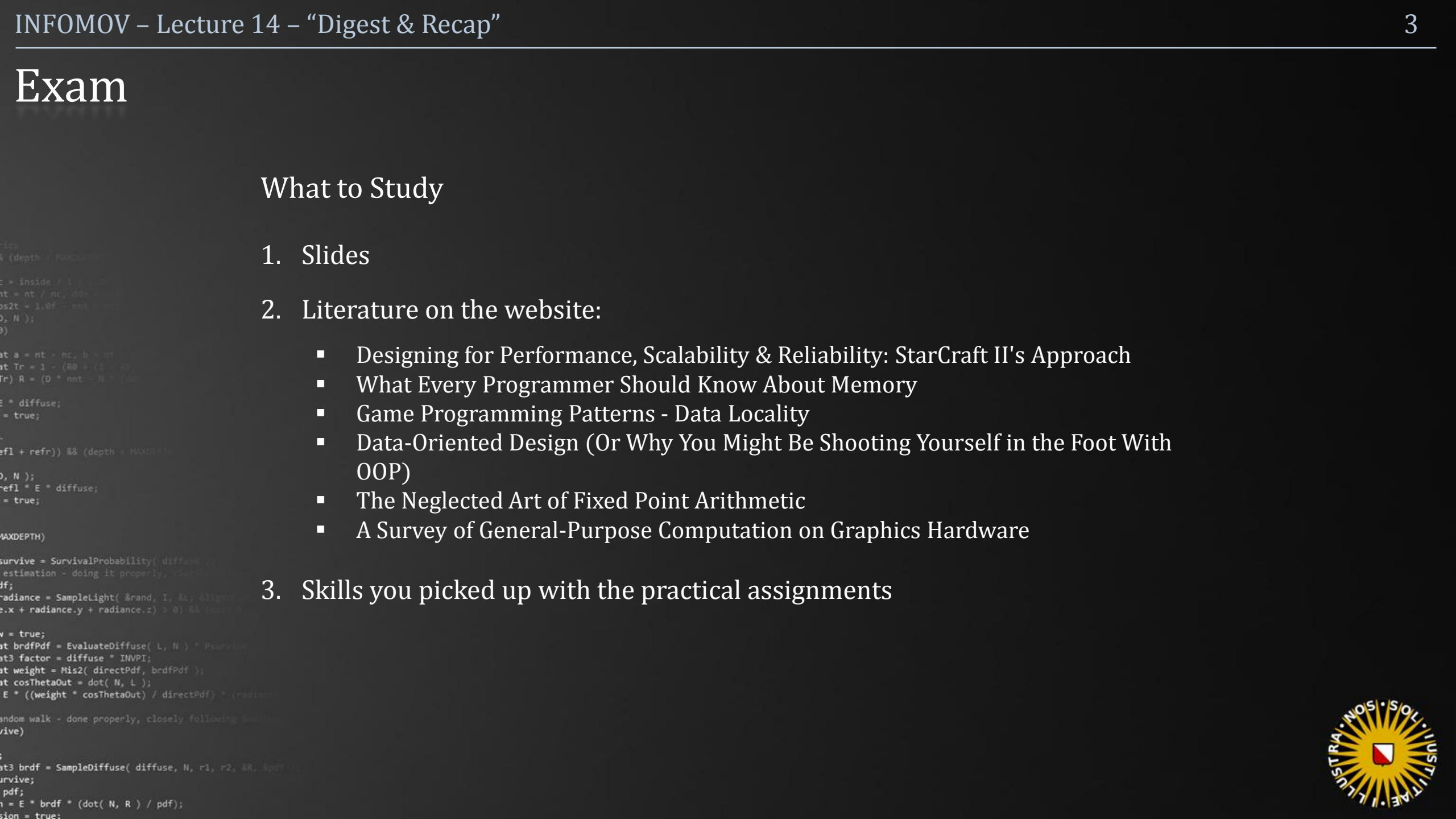
Today’s Agenda:

- Exam
- Digest
- Grand Recap
- Now What

Exam

What to Study

1. Slides
2. Literature on the website:
   - Designing for Performance, Scalability & Reliability: StarCraft II's Approach
   - What Every Programmer Should Know About Memory
   - Game Programming Patterns - Data Locality
   - Data-Oriented Design (Or Why You Might Be Shooting Yourself in the Foot With OOP)
   - The Neglected Art of Fixed Point Arithmetic
   - A Survey of General-Purpose Computation on Graphics Hardware
3. Skills you picked up with the practical assignments



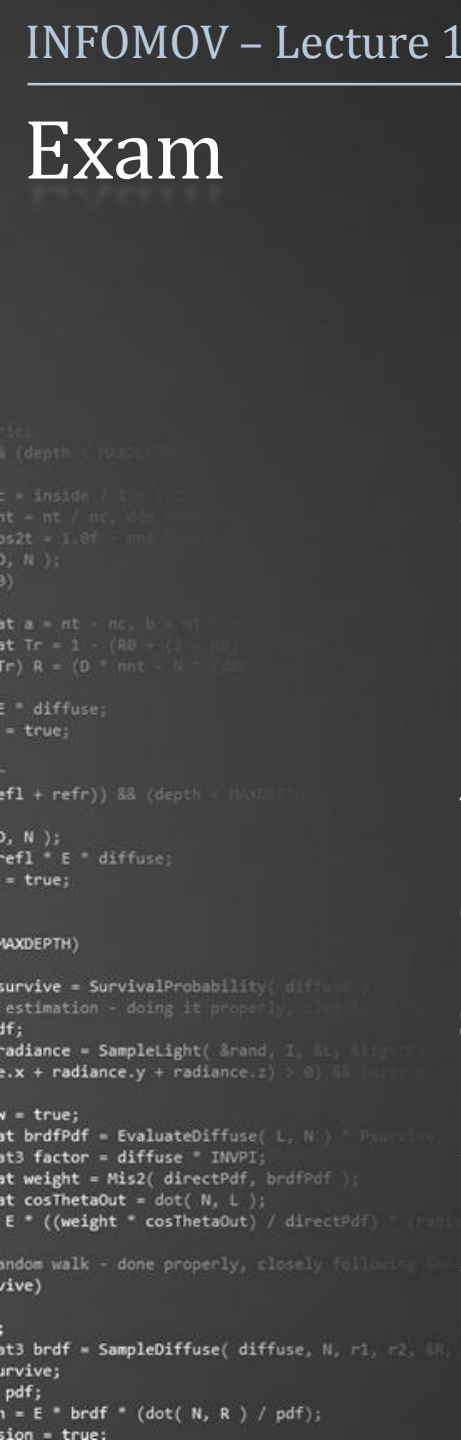
Exam

You may bring a dictionary to the exam.
You may **not** bring notes to the exam.
You may bring pizza to the exam.

Example Questions

1. What is ‘false sharing’?
2. A class that consists of 28 bytes of data is padded to 32 bytes and aligned to a cache line boundary. A large array of objects of this type is then accessed in random order. Why is the padding a bad idea in this case?
3. CPUs and GPUs have fundamentally different core strategies for dealing with latencies such as memory access time. What are these strategies?
4. Why is a 'divide and conquer' algorithm in general unsuitable for GPGPU?
5. What is the purpose of the Northbridge in a modern PC architecture?
6. What is a NUMA architecture?
7. What is ‘hot/cold splitting’?
8. Explain what a super-scalar pipeline is.

Expect a limited number (~10) of open questions covering the main topics: CPU-, cache- and GPU architecture, SIMD, Data Oriented Design, fixed point math.
Some questions explicitly require that you read the literature.
You will have plenty of time to answer the questions.



Today’s Agenda:

- Exam
- Digest
- Grand Recap
- Now What

Digest

Patterns: Vectorization

Optimal use of SIMD: independent lanes in parallel, which naturally extends to 8-wide, 16-wide etc.

Optimal use of GPGPU: large number of independent tasks running in parallel.

Similar pitfalls (conditional code, dependencies / concurrency issues).

Successful algorithm conversion can yield linear speedup in number of lanes.

Digest

Patterns: Vectorization

“The only correct SSE code / GPGPU program is one where many scalar threads run concurrently and independently”

(this pretty much rules out auto-vectorization by the compiler – go manual!)

(this requires suitable data structures: typically SoA)

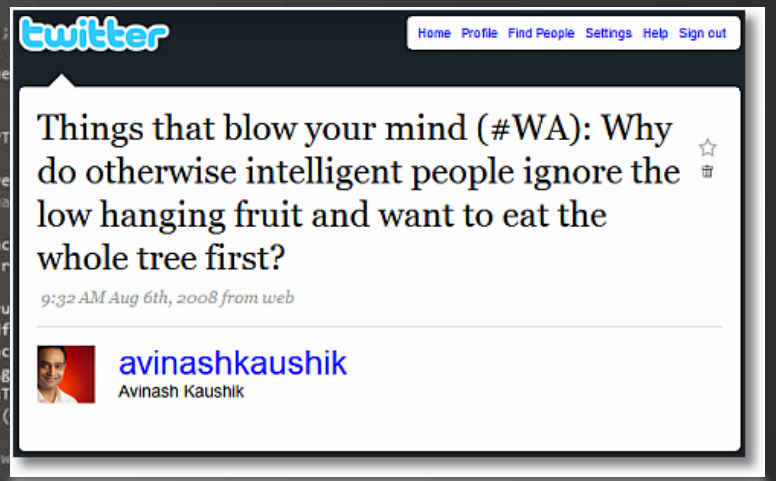
Digest

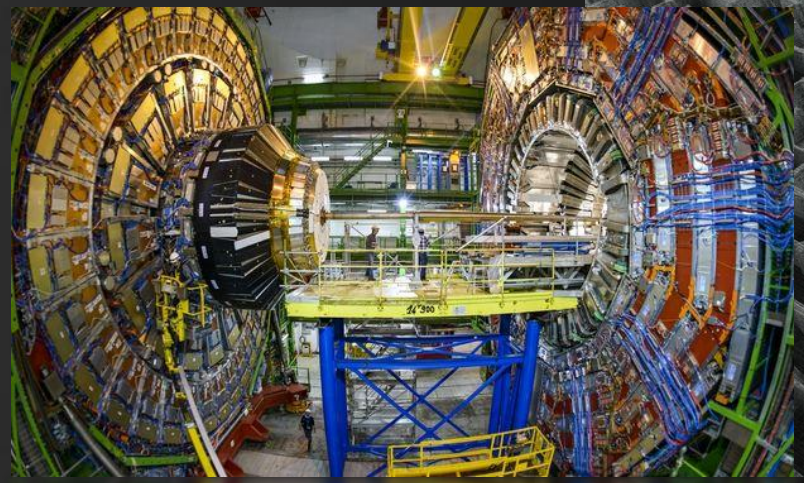
The Relevance of Low Level

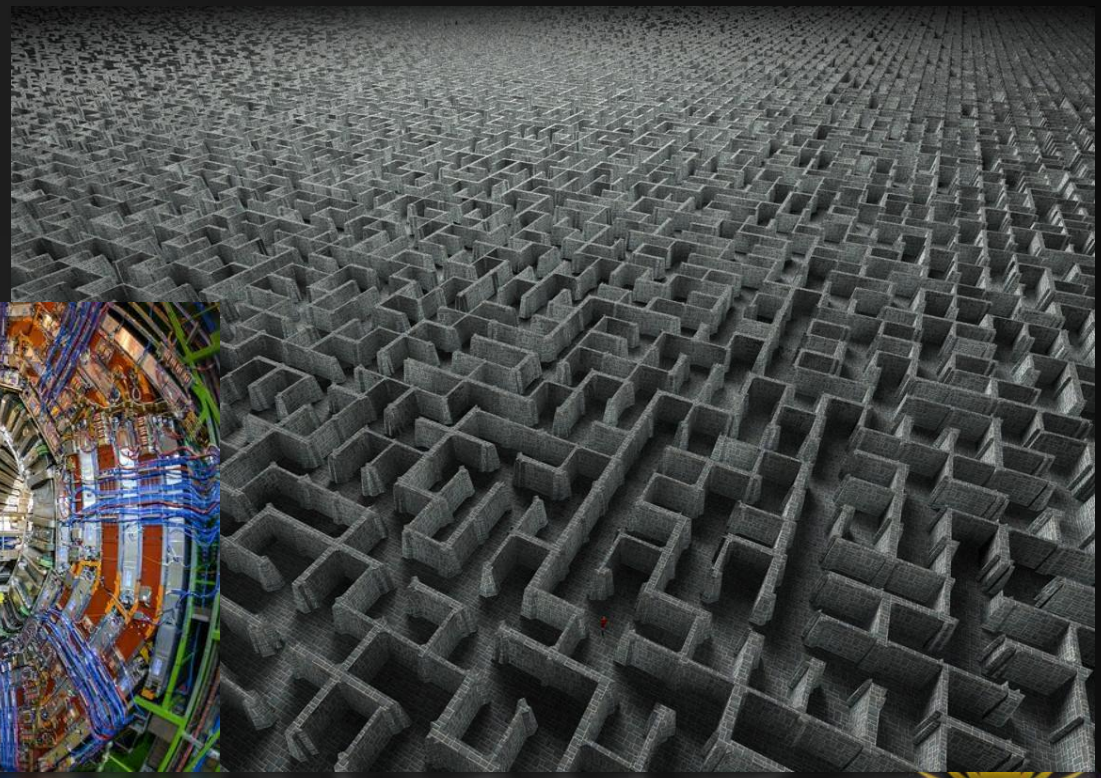
Small gains?

Understanding the hardware

One more percent – Programmer’s Sudoku







Digest

Multi-threading

Considered ‘trivial’ – but it isn’t

Hard to get linear speedup (typical: 2x on 8 cores…)

Increasingly relevant

May affect high level optimization greatly

Benefits from vectorization

Covered in other UU courses, e.g. concurrency (next block, but in bachelor).

Digest

Automatic Optimization

Compilers:

Not all compilers are equal

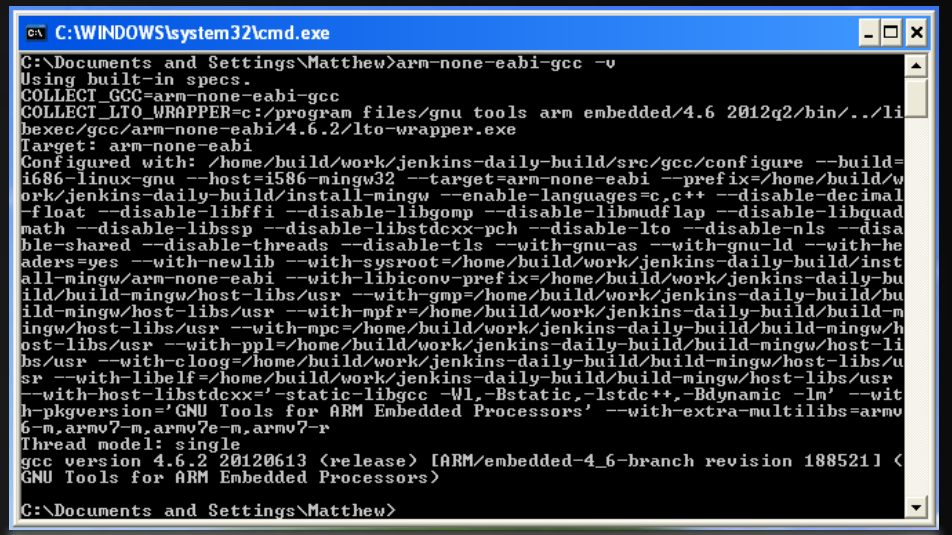
Will do a fair bit of optimization for you

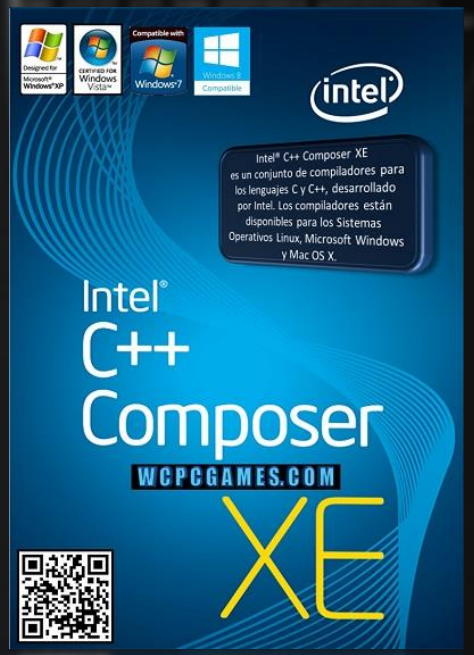
Will tune it to different processors

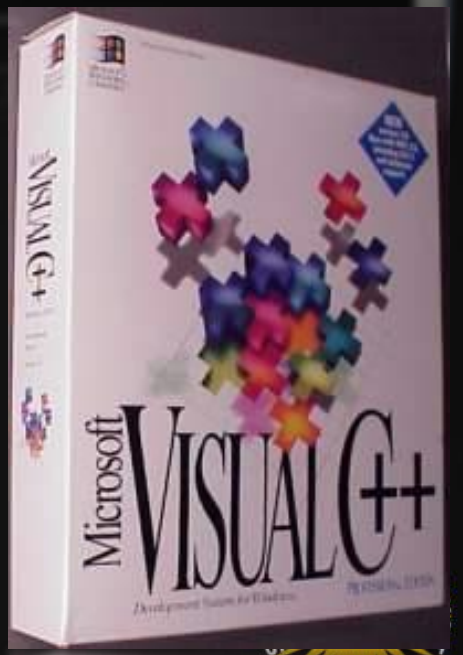
Will sometimes vectorize for you

But: have to be conservative

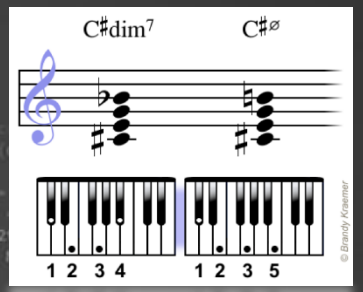
Creating optimizing compilers is a job profile





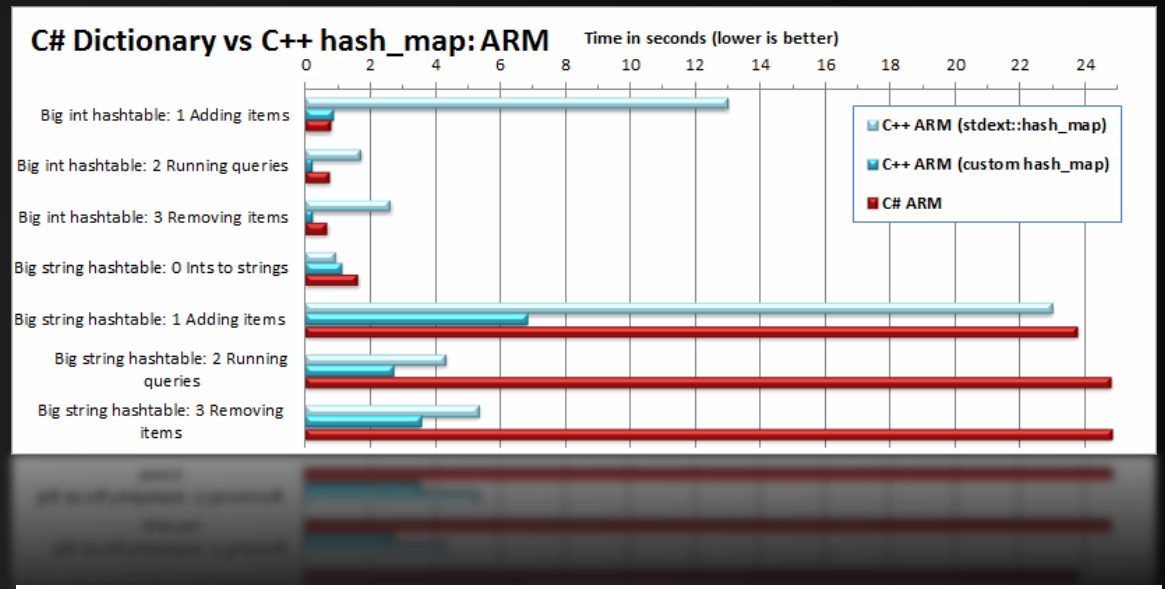


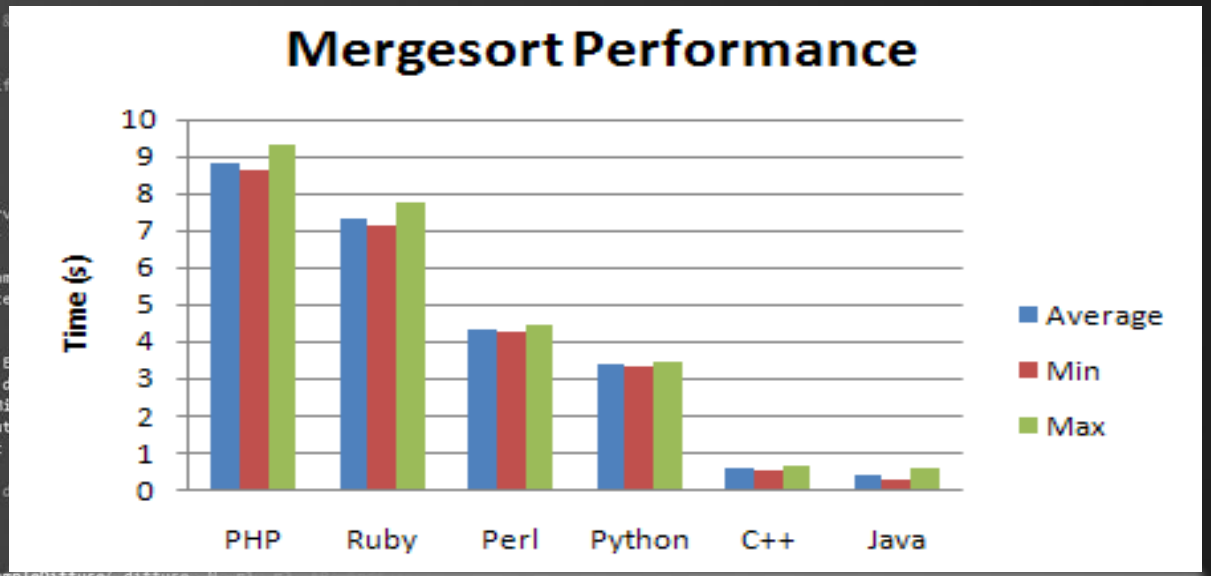
Digest

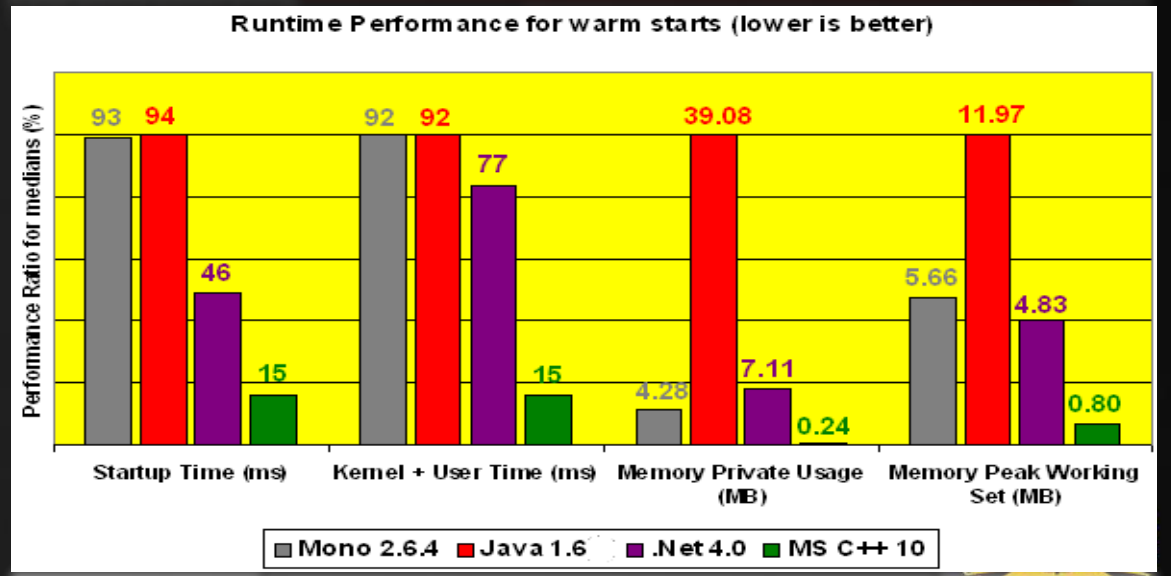


INFOMOV / C#

High level still works
Profiling still works
Some low level still works
Performance Basis: C# versus C++







Digest

Implementation	Lang	sudoku:t	matmul:t	matmul:m
ICC-12.0.3	C	1.0	1.8	31.8
Clang@LLVM-2.9	C	1.0	2.3	31.7
GCC-4.3.2	C	1.0	2.3	31.7
C#@Mono-2.10.1	C#	3.8	8.9	40.6
GDC-0.24	D	1.1	3.3	33.9
LDC-20110428@LLVM-2.9	D	1.1	2.4	31.4
6g-20110424	Go	2.3	3.1	38.2
Java@JRE-1.6.0_25	Java	1.7	2.6	67.1
V8-r8384	Javascript	3.7	2.6	141.6
JaegarMonkey-a95d42642281	Javascript	18.1	16.4	35.8
LuaJIT-2.0.1 (JIT-on)	Lua	3.7	2.5	33.2
Lua-5.1.4	Lua	50.5	68.3	65.4
Perl-5.12.2	Perl	121.2	230.3	225.6
Shedskin-0.9@GCC-4.3.2	Python	4.4	3.7	50.4
IronPython-2.7@Mono-2.10.1	Python	100.9	202.7	190.2
Jython-2.5.2@JRE-1.6.0_25	Python	136.3	731.4	355.6
PyPy-1.4.1	Python	19.5	8.5	84.1
R-2.13.0	R	999.9	1736.3	57.2
JRuby-1.6.1@JRE-1.6.0_25	Ruby	71.1	238.2	342.5
Rubinius-1.2.3	Ruby	135.5	298.1	162.5

sudoku:t: time for solving 20 extremely hard Sudoku’s 50 times.

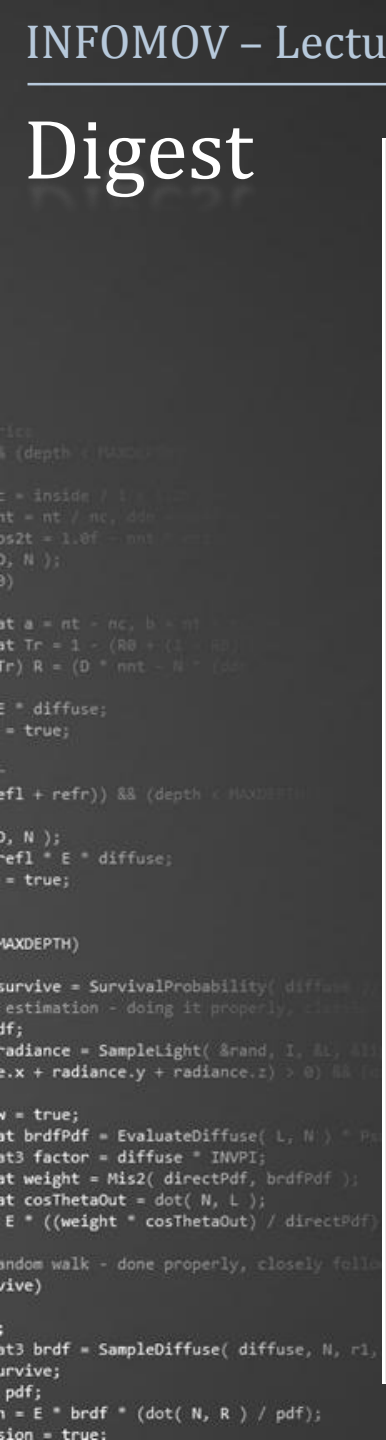
matmul:t: time (relative to ICC) for multiplying two 1000x1000 matrices (standard $O(N^2)$ algorithm).

matmul:m: memory (in megabytes) for multiplying two 1000x1000 matrices.

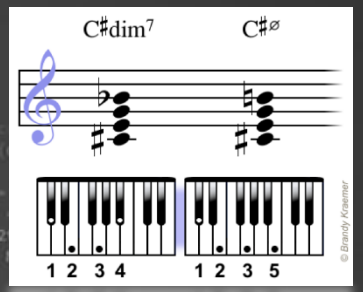
Reference:

Intel C++ compiler version 12.0.3, ‘10;

Java JRE: End of 2011;
Mono 2.1: End of 2010.



Digest



INFOMOV / C#

High level still works
Profiling still works
Some low level still works
Performance Basis: C# versus C++

C#-specific optimization:
http://www.dotnetperls.com/optimization
https://www.udemy.com/csharp-performance-tricks-how-to-radically-optimize-your-code/
http://www.c-sharpcorner.com/UploadFile/47fc0a/code-optimization-techniques/

Digest

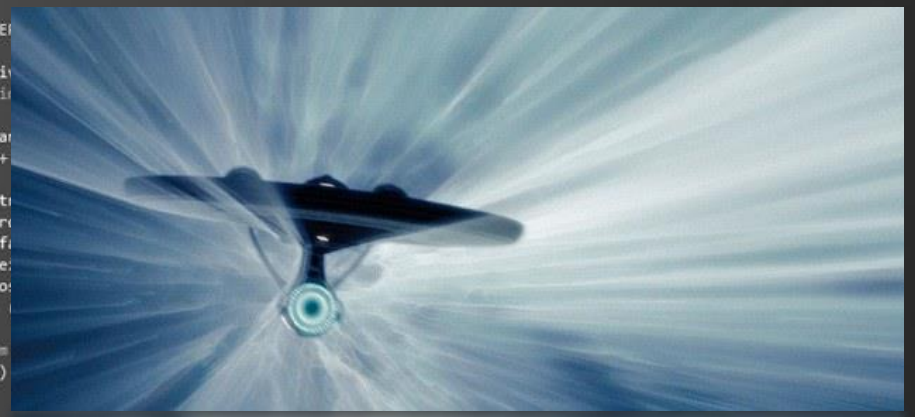
The Process

10x and more – proven? (did we use realistic scenarios?)

Counter-intuitive steps – attractive square roots

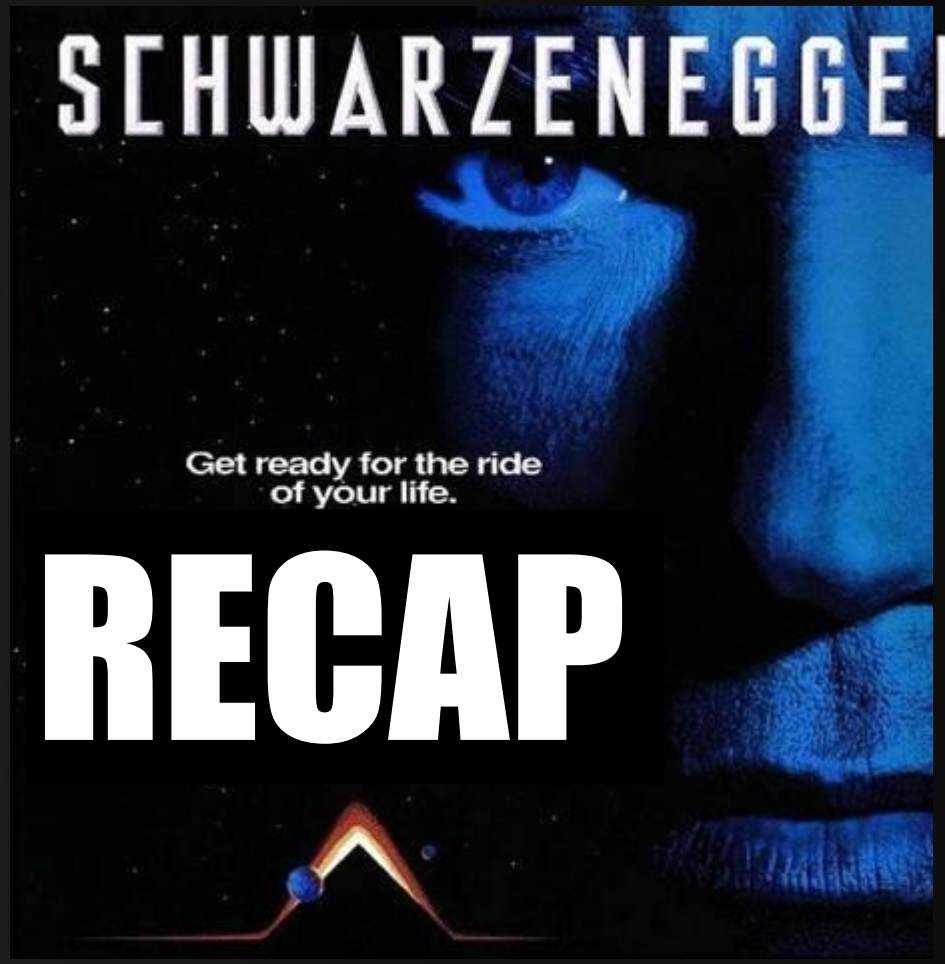
Importance of profiling

Is the process generic?



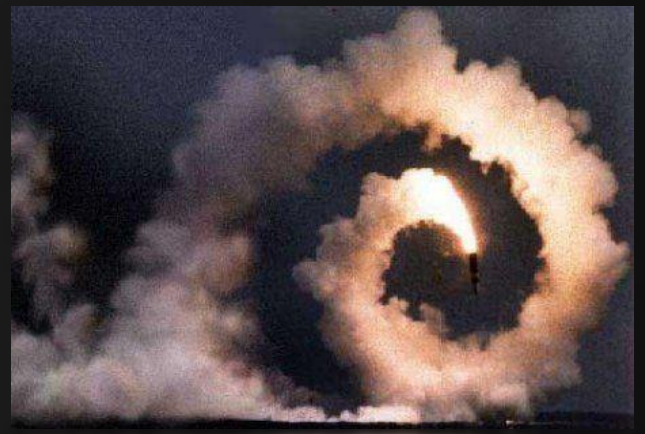
Today’s Agenda:

- Exam
- Digest
- Grand Recap
- Now What

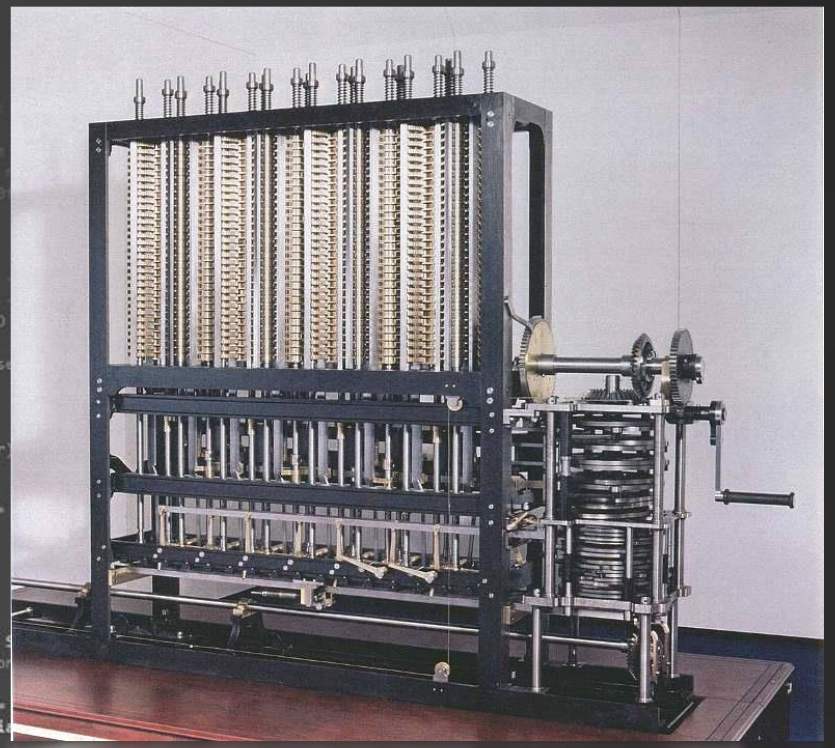


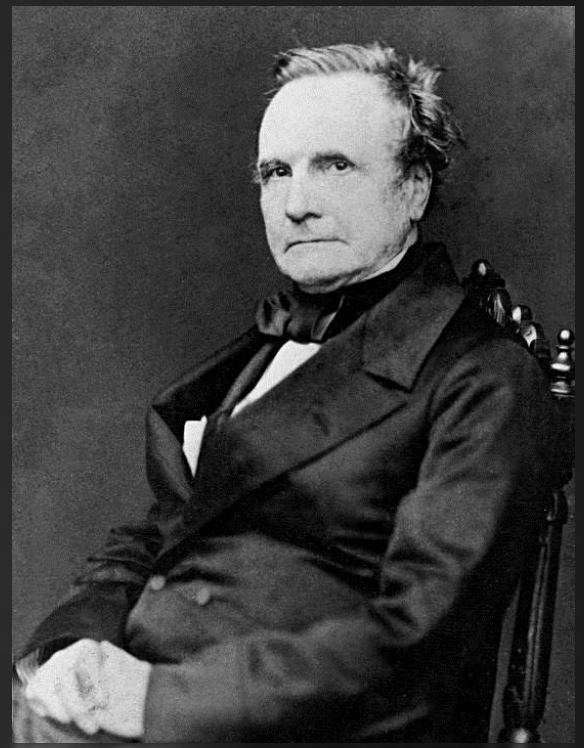
TOTAL RECAP

Recap

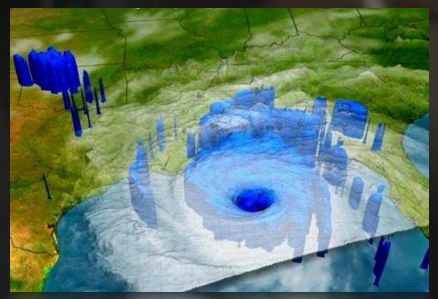


Recap – lecture 1

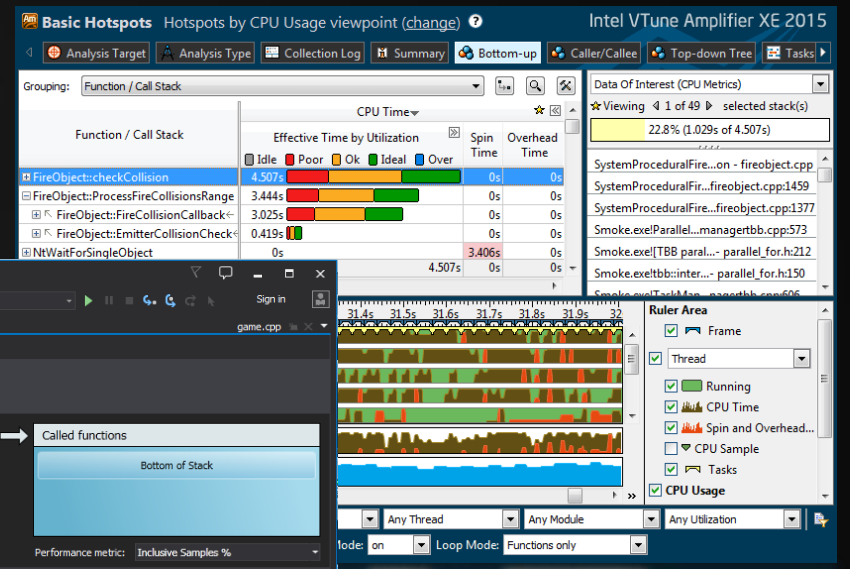


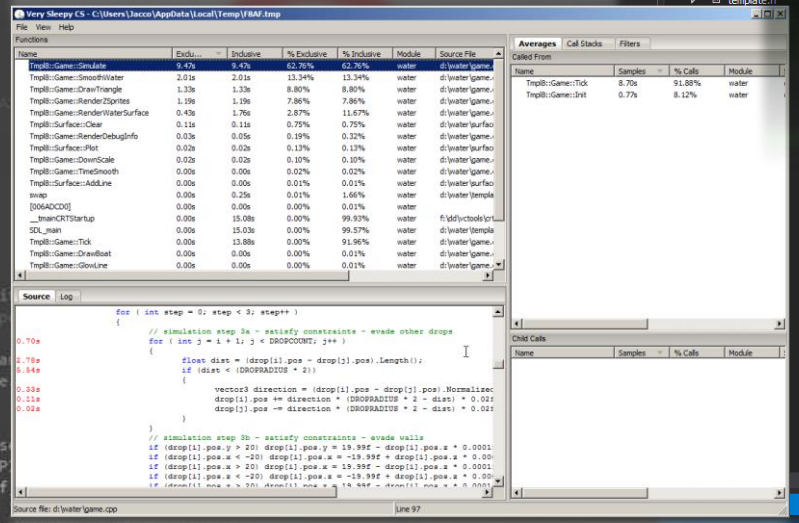


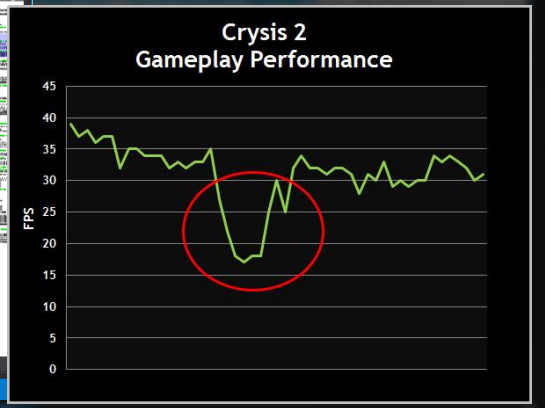
- Profiling
- High Level
- Basic Low Level
- Cache & Memory
- Data-centric
- Compilers
- Fixed-point Arithmetic
- CPU architecture
- **SIMD**
- GPGPU



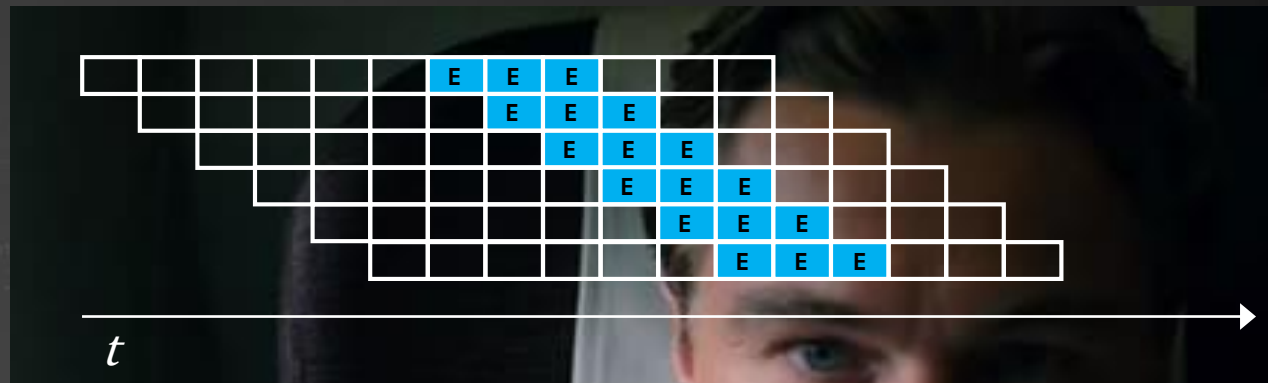
Recap – lecture 1







Recap – lecture 2



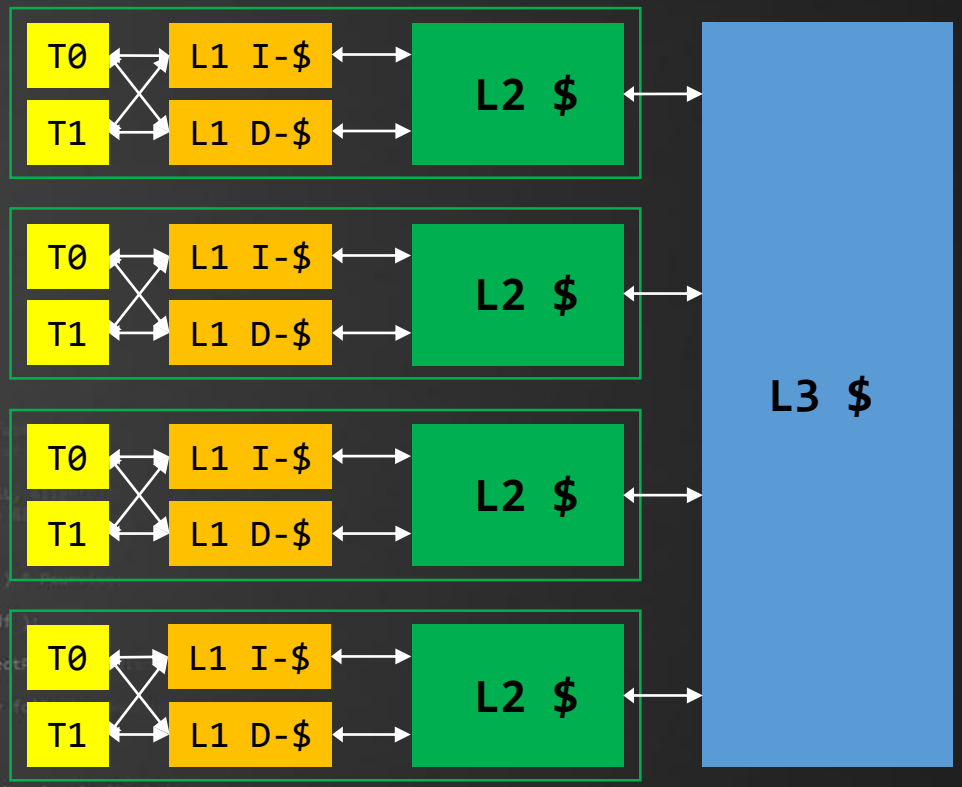
Red = u4 & (255 << 16);
Green = u4 & (255 << 8);
Blue = u4 & 255;

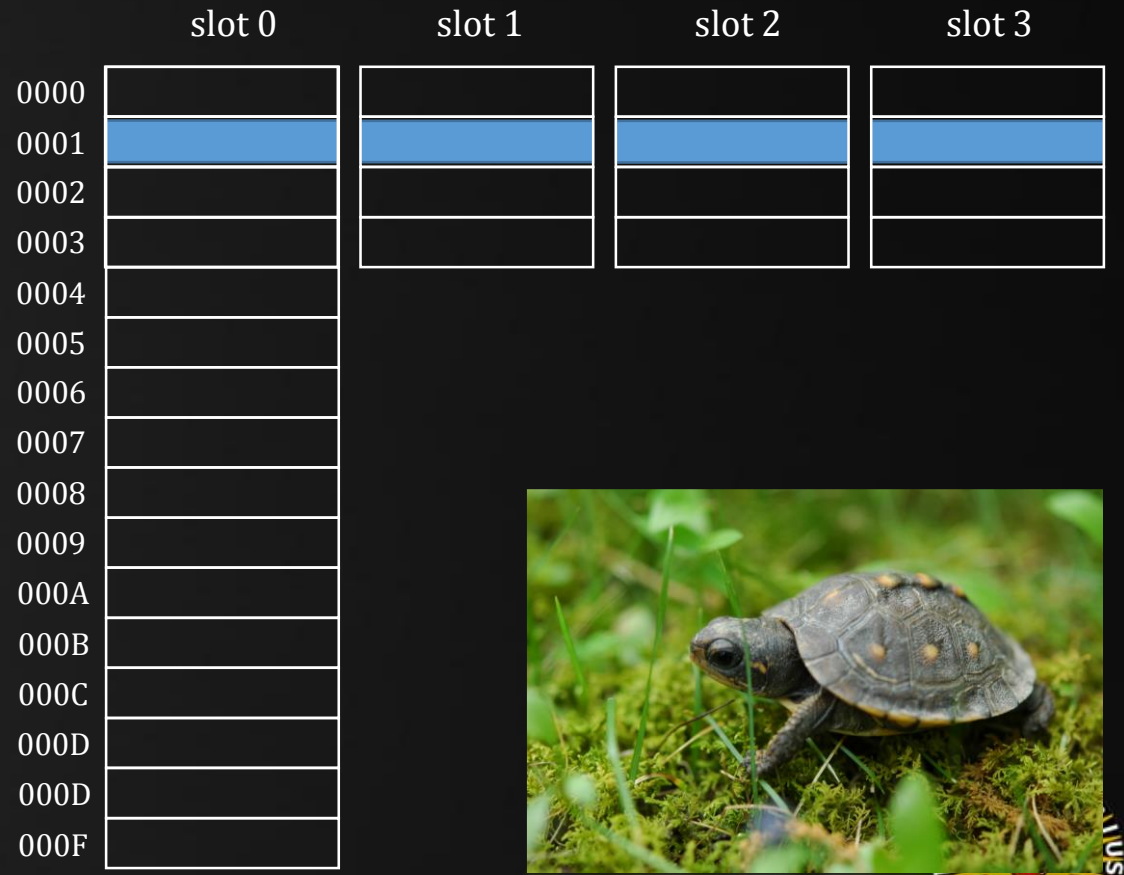

```
fldz
xor ecx, ecx
fld dword ptr ds:[405290h]
mov edx, 28929227h
fld dword ptr ds:[40528Ch]
push esi
mov esi, 0C350h      = 50000
```

```
add ecx, edx
mov eax, 91D2A969h
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st

mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1),st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Fh
dec esi
jne tobetimed<0>+1Fh
```

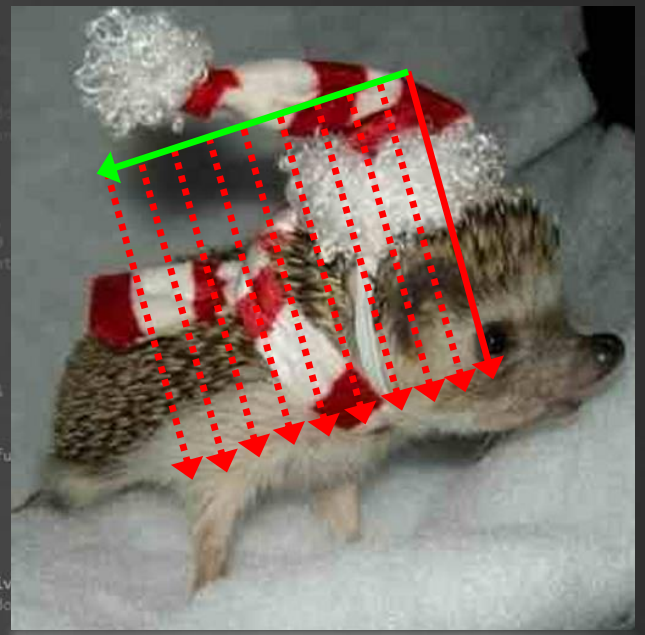
$= \frac{2^{46}}{28763}$ (!!)

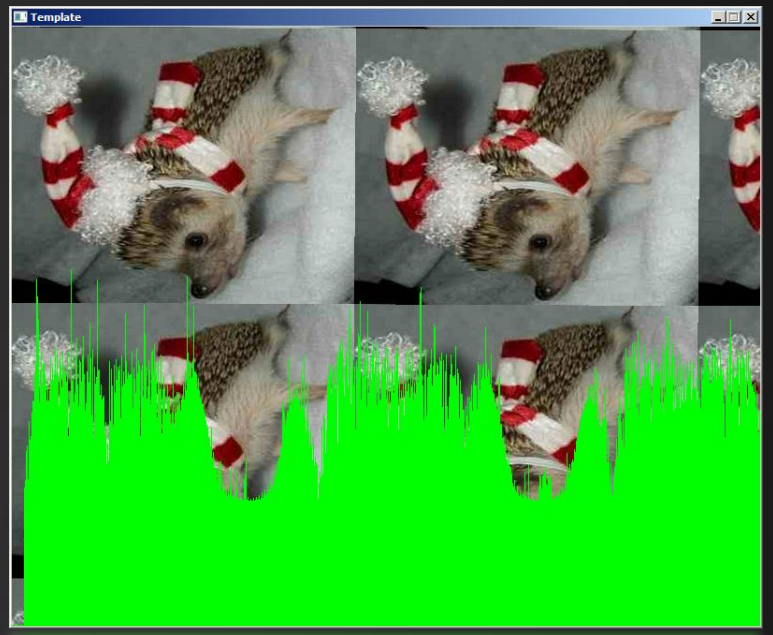
Recap – lecture 3

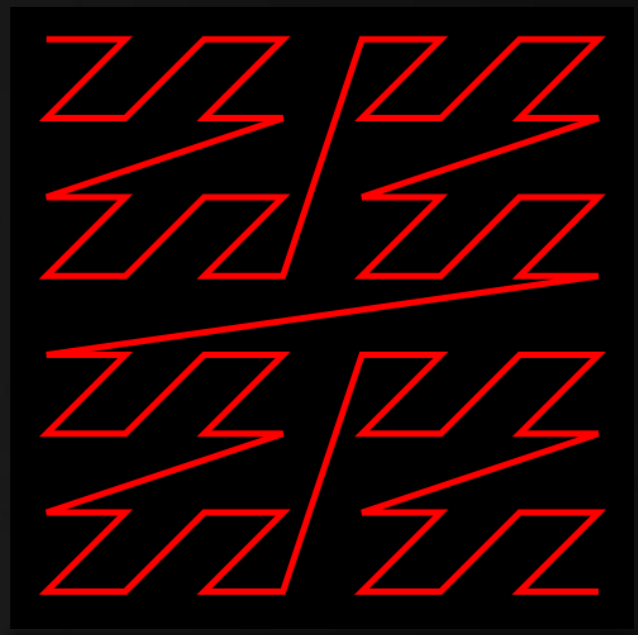




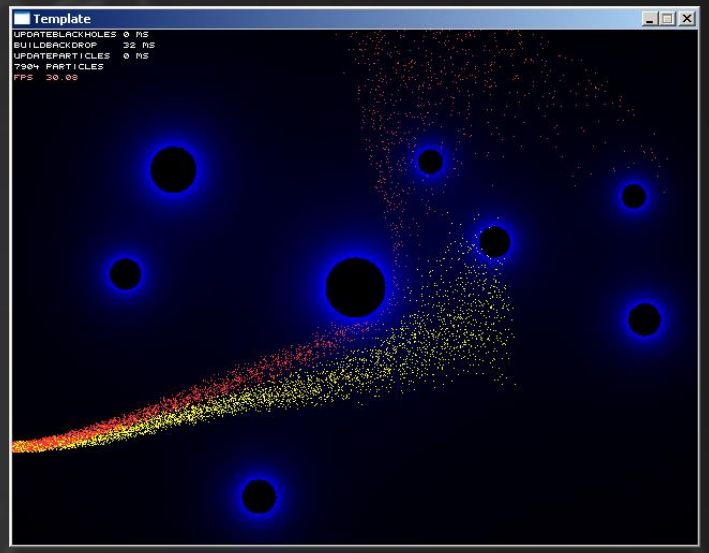
Recap – lecture 5

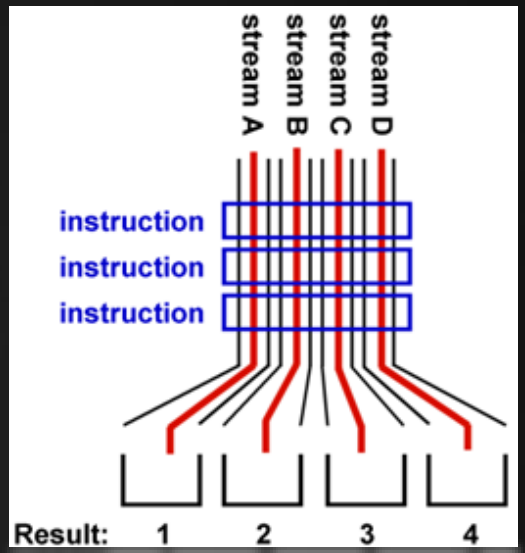






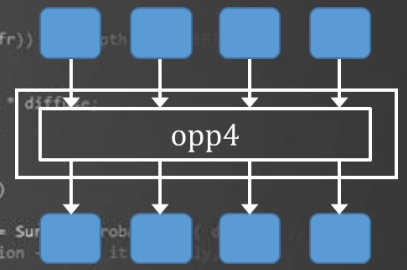
Recap – lecture 6 & 7





Agner Fog:
“Automatic vectorization is the easiest way of generating SIMD code, and I would recommend to use this method when it works. Automatic vectorization may fail or produce suboptimal code in the following cases:

- when the algorithm is too complex.
- when data have to be re-arranged in order to fit into vectors and it is not obvious to the compiler how to do this or when other parts of the code needs to be changed to handle the re-arranged data.
- when it is not known to the compiler which data sets are bigger or smaller than the vector size.
- when it is not known to the compiler whether the size of a data set is a multiple of the vector size or not.
- when the algorithm involves calls to functions that are defined elsewhere or cannot be inlined and which are not readily available in vector versions.
- when the algorithm involves many branches that are not easily vectorized.
- when floating point operations have to be reordered or transformed and it is not known to the compiler whether these transformations are permissible with respect to precision, overflow, etc.
- when functions are implemented with lookup tables.



AoS

SoA

SIMD Basics

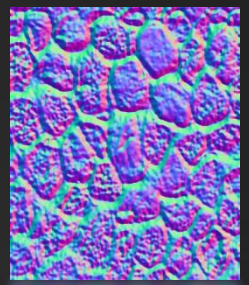
Other instructions:

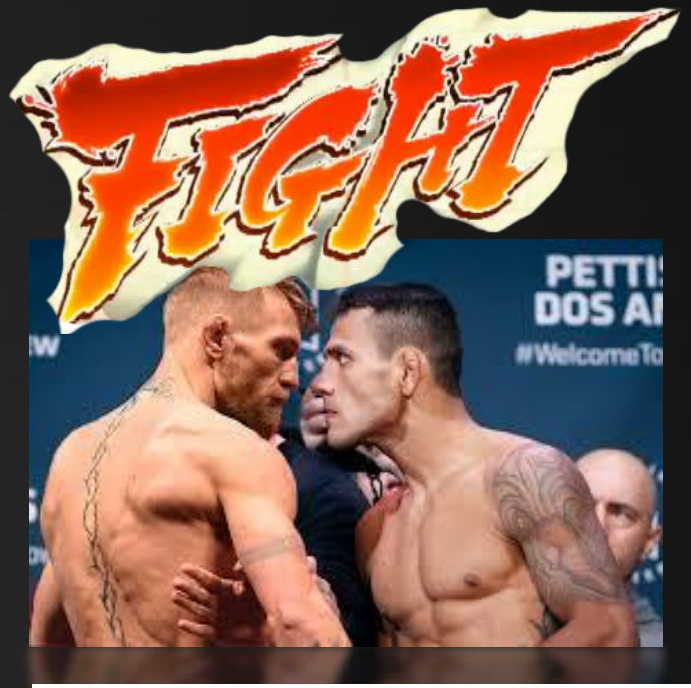
```
__m128 c4 = _mm_div_ps( a4, b4 ); // component-wise division
__m128 d4 = _mm_sqrt_ps( a4 );    // four square roots
__m128 d4 = _mm_rcp_ps( a4 );     // four reciprocals
__m128 d4 = _mm_rsqrt_ps( a4 );   // four reciprocal square roots (!)

__m128 d4 = _mm_max_ps( a4, b4 );
__m128 d4 = _mm_min_ps( a4, b4 );
```

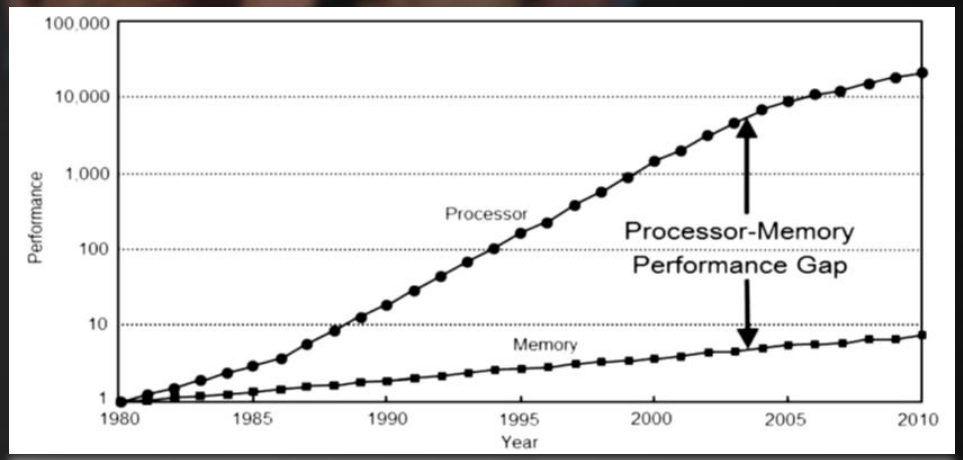
Keep the assembler-like syntax in mind:

```
__m128 d4 = dx4 * dx4 + dy4 * dy4;
```

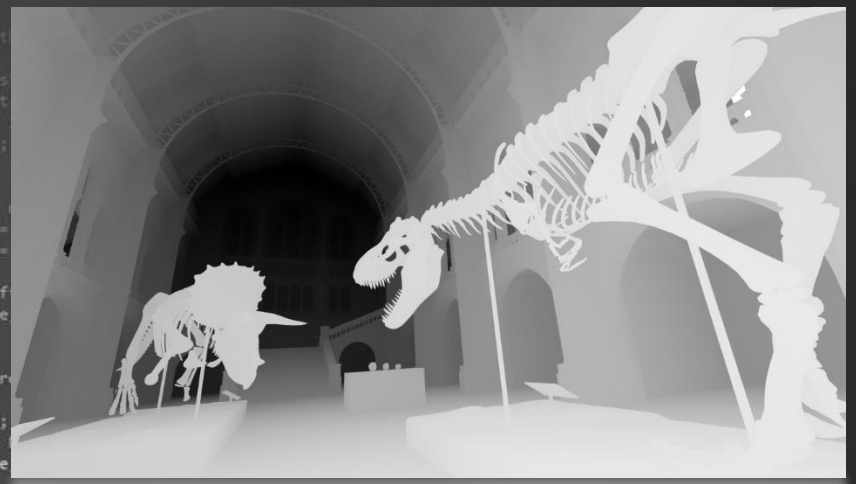
Recap – lecture 8

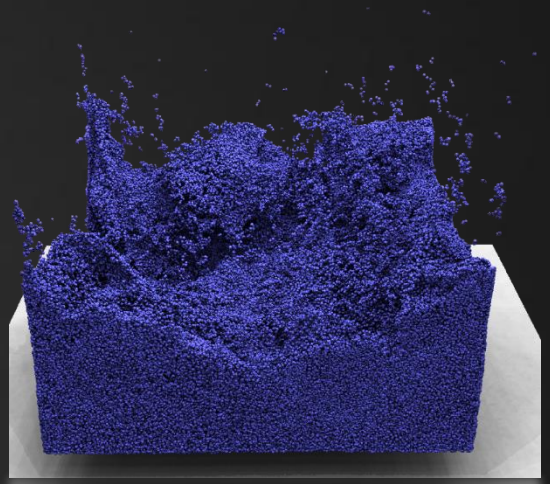


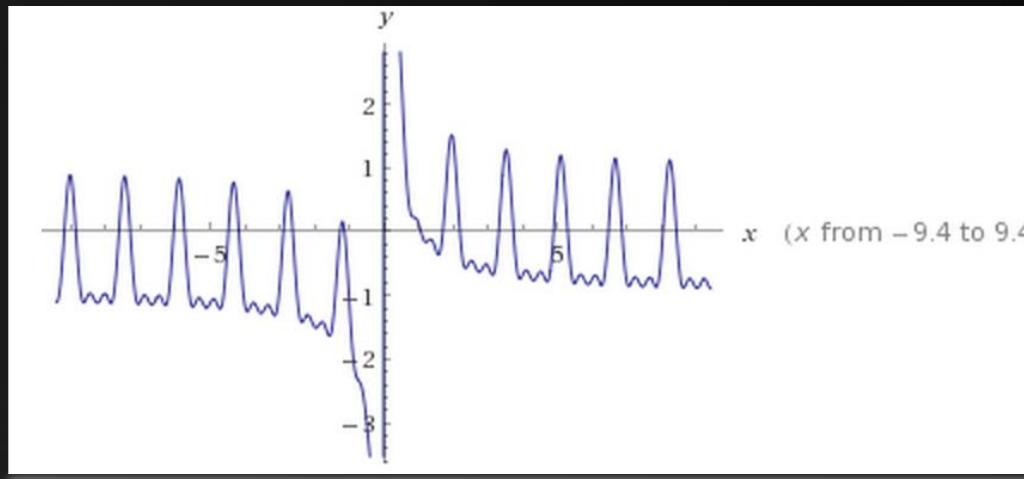


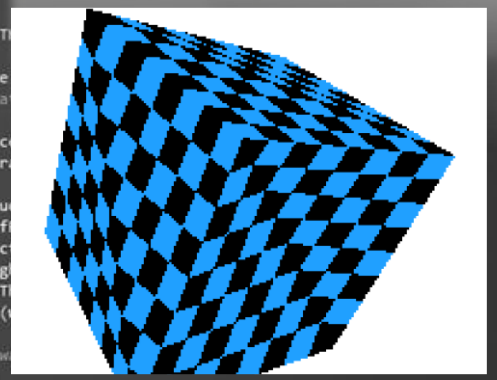


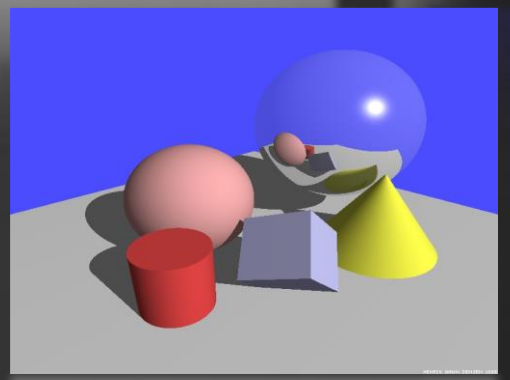
Recap – lecture 9












```
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
urvive;
 pdf;
 = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

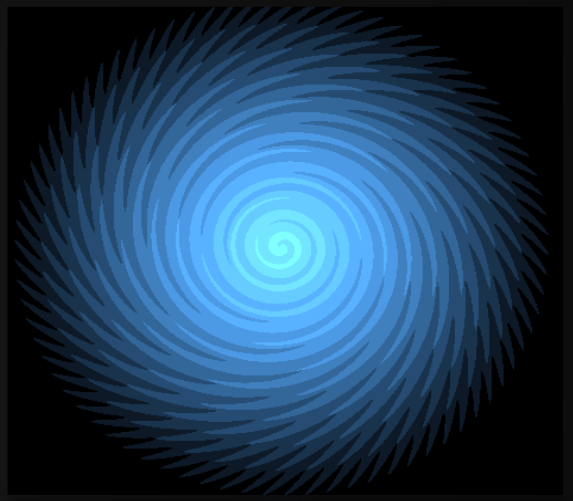
Recap – lecture 10

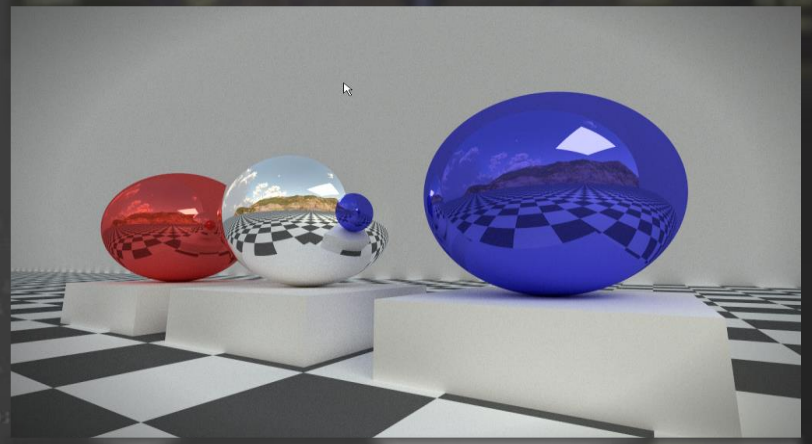
POTATO SALAD

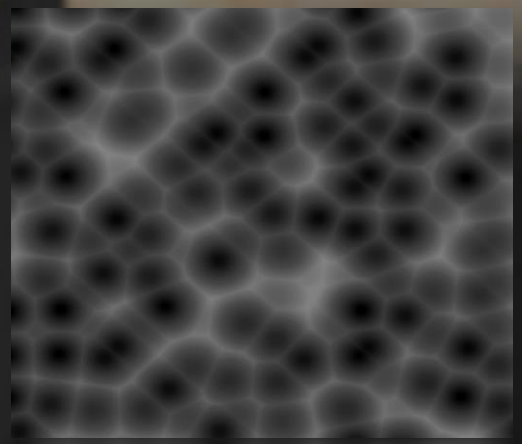
Some good cooks sprinkle grated pimiento cheese on this

- 4 cups diced cooked potatoes
- 1 cup sliced celery
- 3 hard-cooked eggs, cut up
- ½ cup finely cut onion or sliced green onions
- ¼ cup sliced radishes
- 1 cup mayonnaise
- 1 tablespoon vinegar
- 1 teaspoon prepared mustard
- 1½ to 2 teaspoons salt
- ⅛ teaspoon pepper
- Lettuce

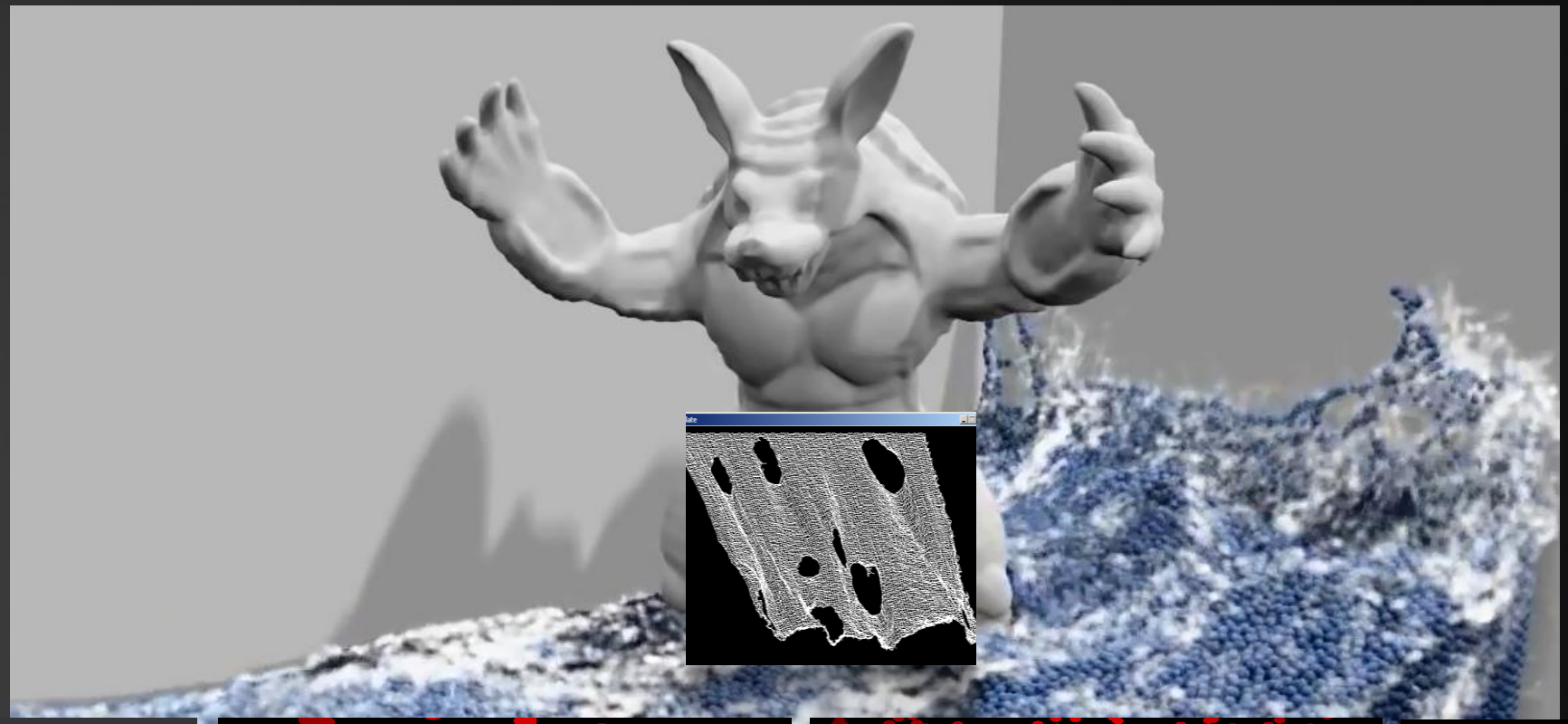
Mix all the ingredients in a bowl. Cover and refrigerate several hours so flavors can blend. Serve on crisp lettuce. Makes 6 servings.

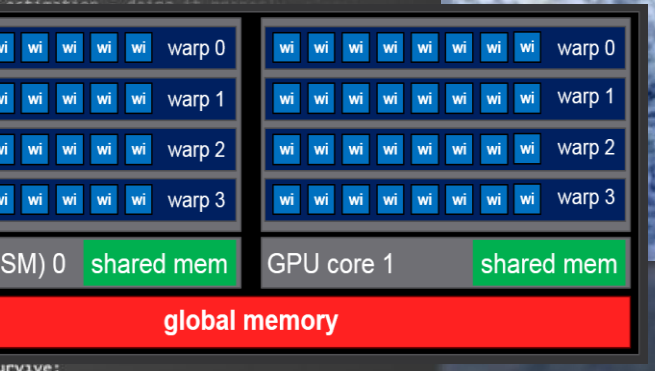


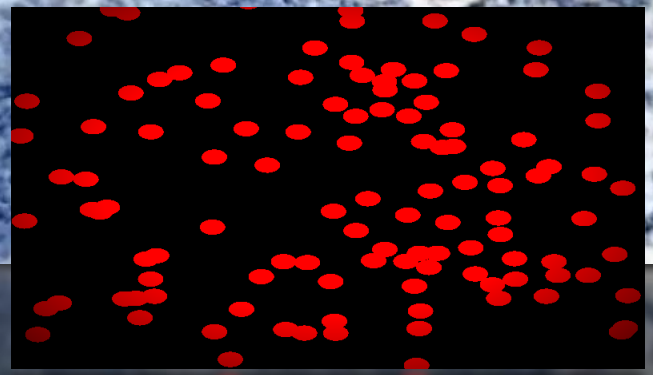


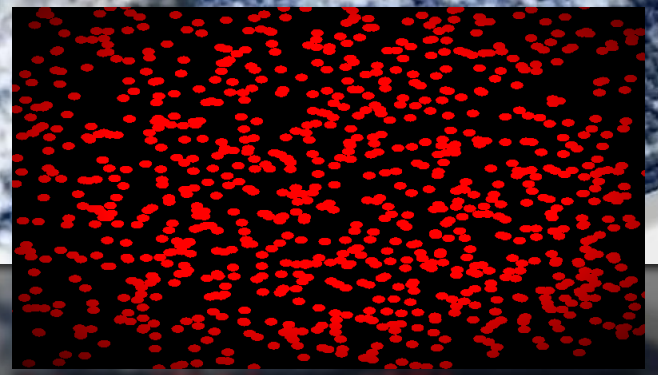


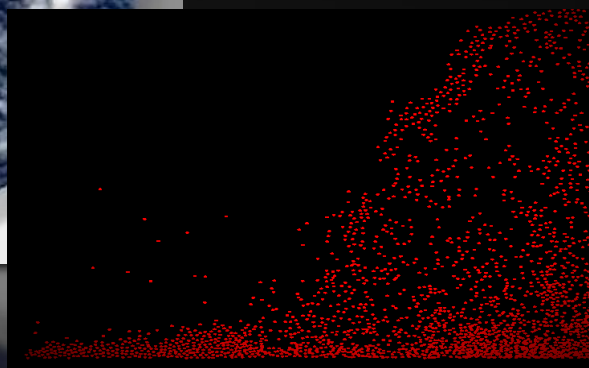
Recap – lecture 11



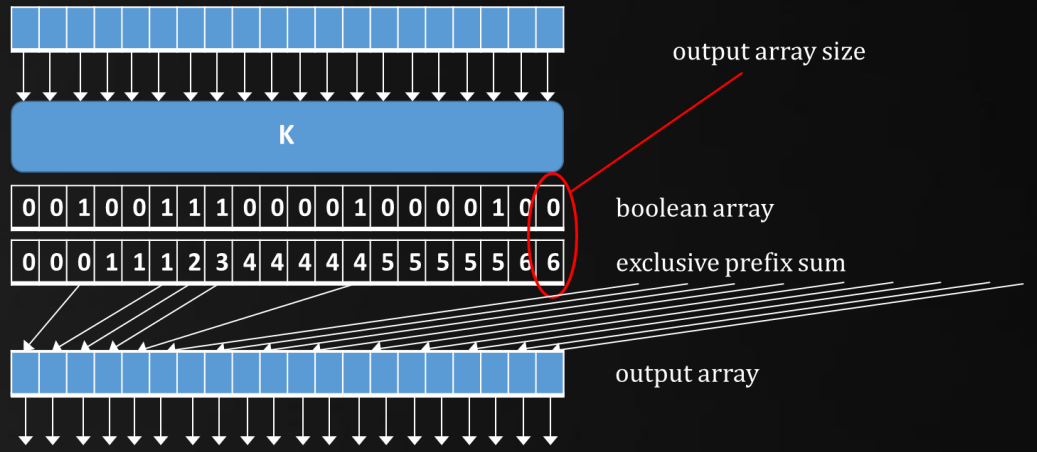


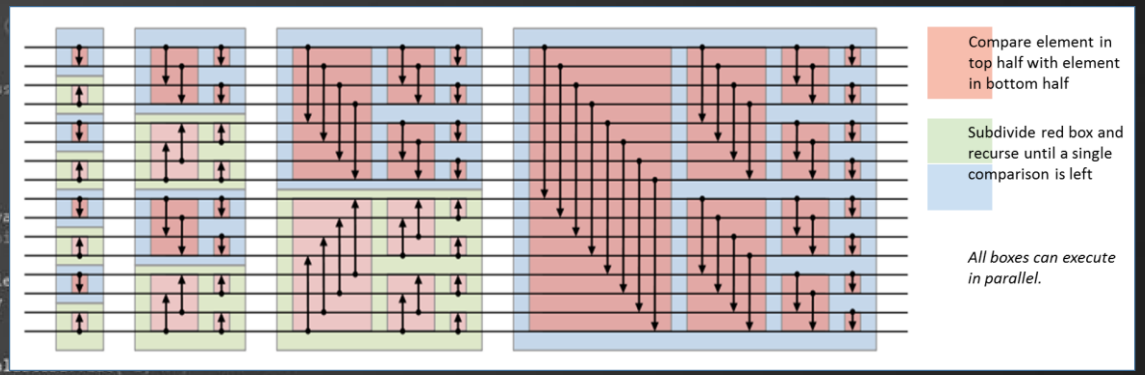


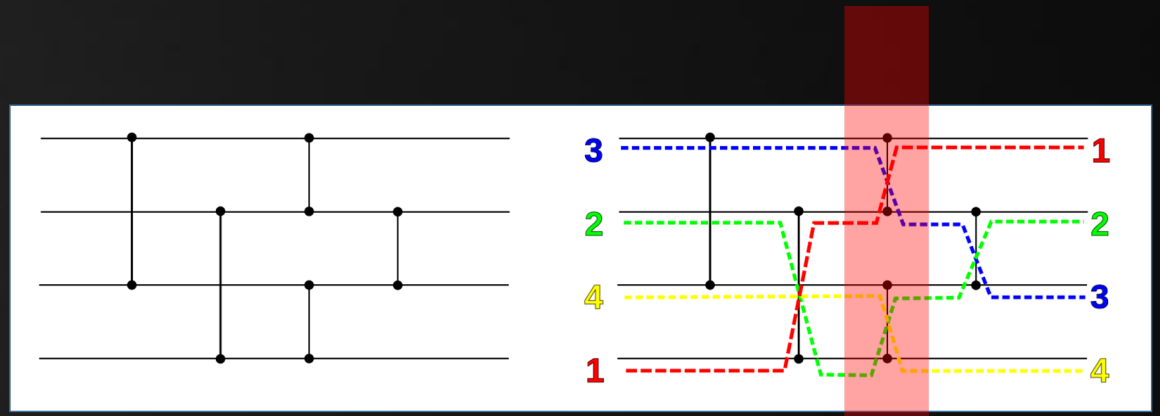




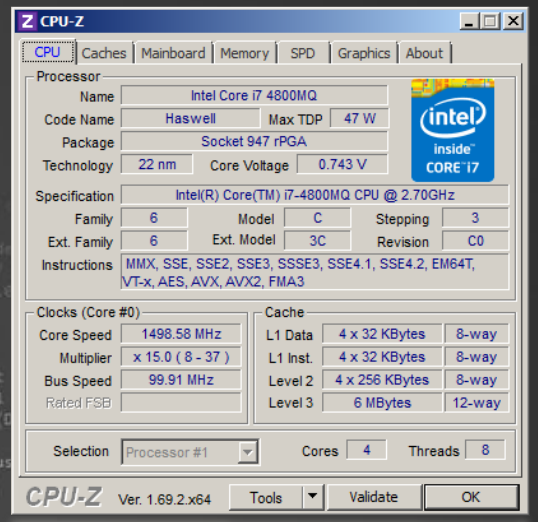
Recap – lecture 12

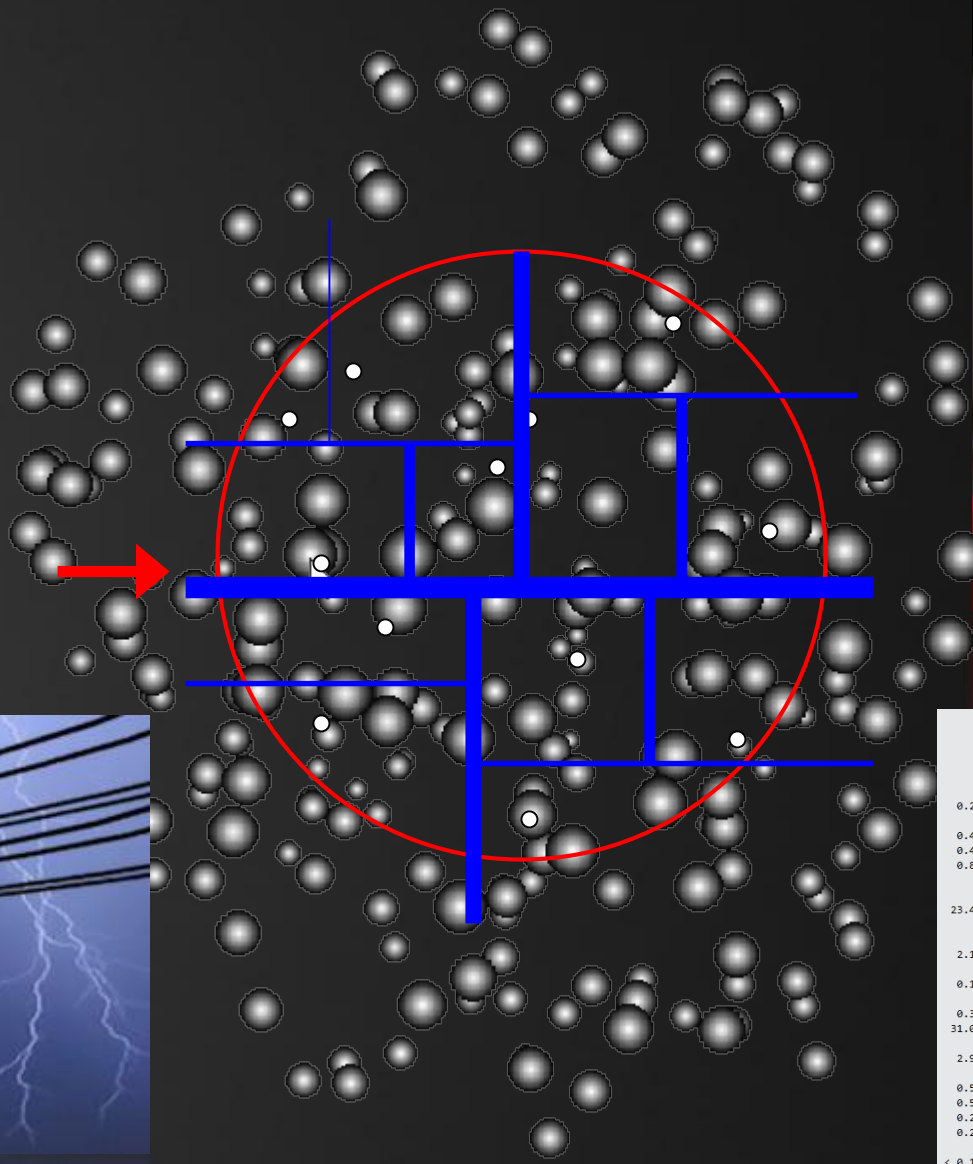






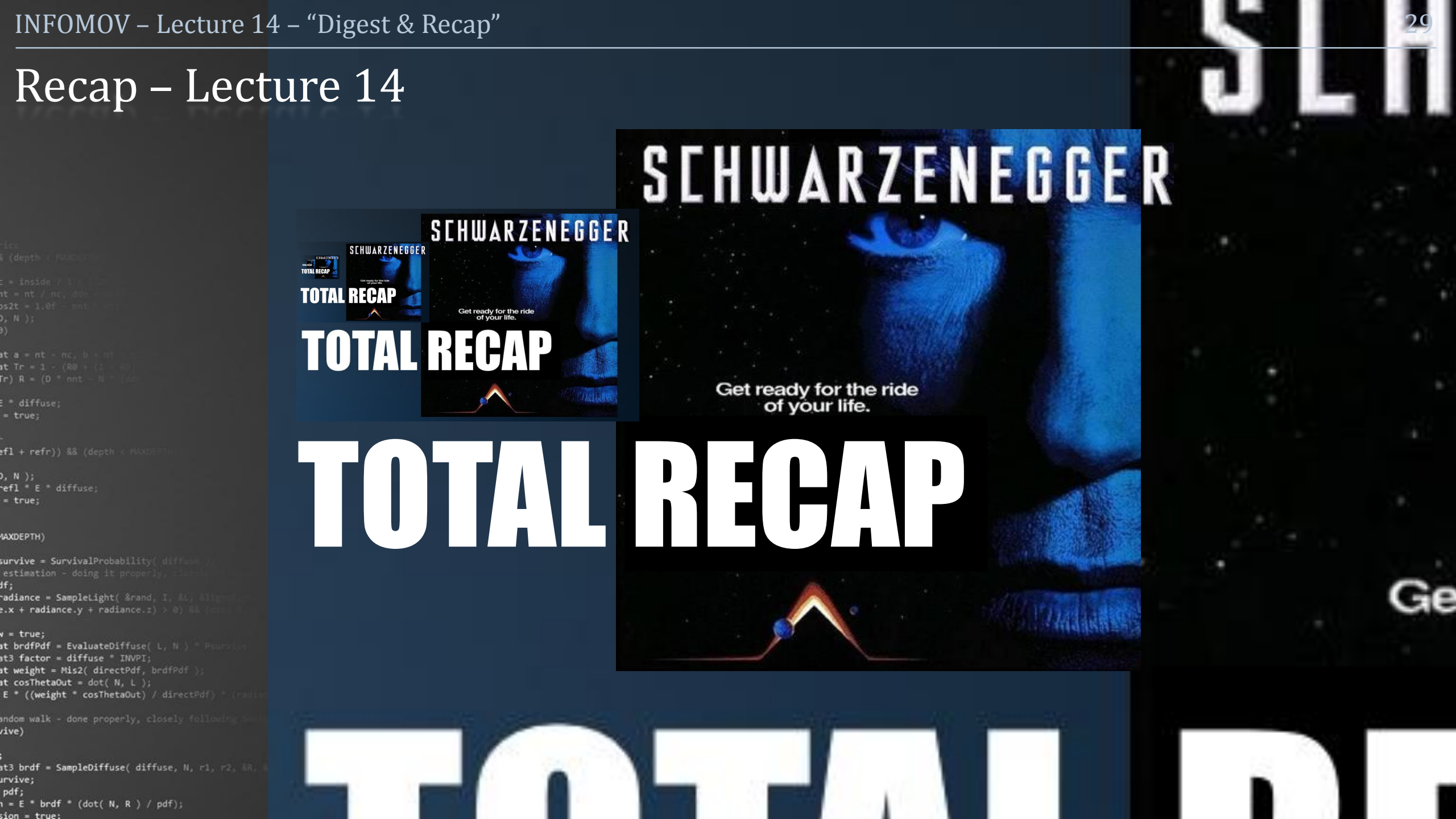
Recap – lecture 13





```
void Game::Tick( float _DT )
{
    // fill the grid
    for( int q = 0; q < 20; q++ )
    {
        memset( nr, 0, SCRWIDTH / 16 * SCRHEIGHT / 16 * 4 );
  0.2 %	for( int i = 0; i < BALLCOUNT; i++ )
        {
  0.4 %		int gx = CLAMP( (int)(pos[i].x / 16 ), 0, SCRWIDTH / 16 - 1 );
  0.4 %		int gy = CLAMP( (int)(pos[i].y / 16 ), 0, SCRHEIGHT / 16 - 1 );
  0.8 %		grid[gy][gx][nr[gy][gx]++] = i;
        }

 23.4 %	screen->Clear( 0 );
        timer t;
        t.reset();
  2.1 %	for( int v = 0; v < SCRHEIGHT / 16; v++ ) for( int u = 0; u < SCRWIDTH / 16; u++ ) for( int j = 0; j < nr[v][u]; j++ )
        {
  0.1 %		int i = grid[v][u][j];
            // draw ball
  0.3 %		if (i == 0) green->Draw( (int)pos[i].x, (int)pos[i].y, screen );
 31.0 %			else ball->Draw( (int)pos[i].x, (int)pos[i].y, screen );
            // update ball position
  2.9 %		pos[i] += vel[i] * speed[i] * 0.05f;
            // screen boundary collisions
  0.5 %		if (pos[i].x < 10) vel[i].x = -vel[i].x, pos[i].x = 10;
  0.5 %		if (pos[i].y < 10) vel[i].y = -vel[i].y, pos[i].y = 10;
  0.2 %		if (pos[i].x > (SCRWIDTH - 20)) vel[i].x = -vel[i].x, pos[i].x = SCRWIDTH - 20;
  0.2 %		if (pos[i].y > (SCRHEIGHT - 20)) vel[i].y = -vel[i].y, pos[i].y = SCRHEIGHT - 20;
            // ball collisions
< 0.1 %		int gx1 = MAX( 0, u - 1 ), gx2 = MIN( SCRWIDTH / 16 - 1, u + 1 );
  0.3 %		int gy1 = MAX( 0, v - 1 ), gy2 = MIN( SCRHEIGHT / 16 - 1, v + 1 );
 11.6 %		for( int y = gy1; y <= gy2; y++ ) for( int x = gx1; x <= gx2; x++ ) for( int k = 0; k < nr[y][x]; k++ )
            {
```

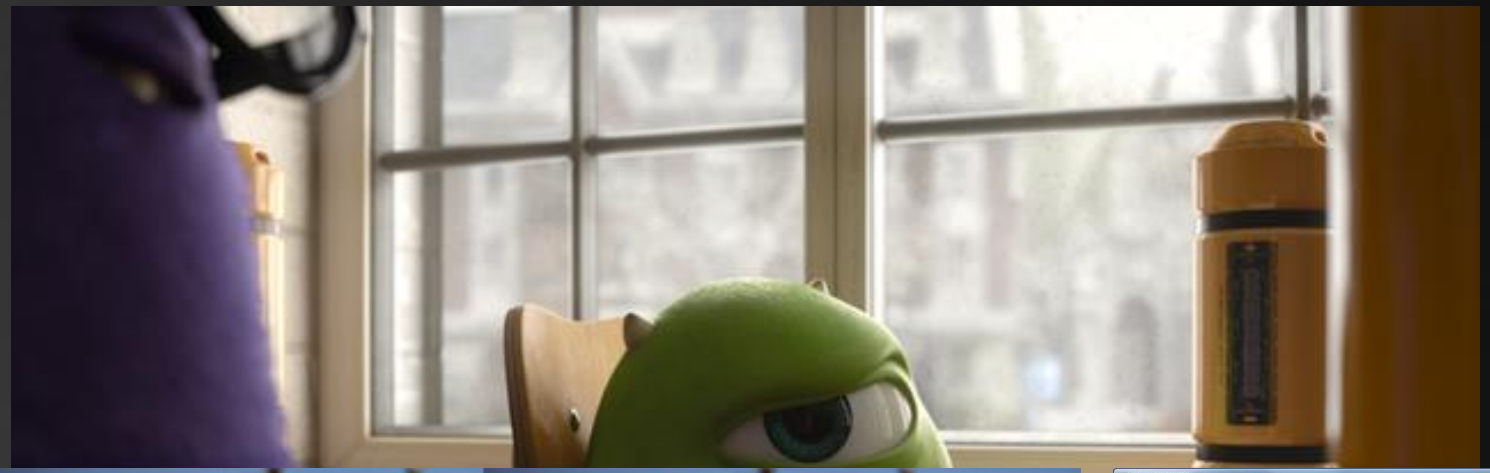
Recap – Lecture 14

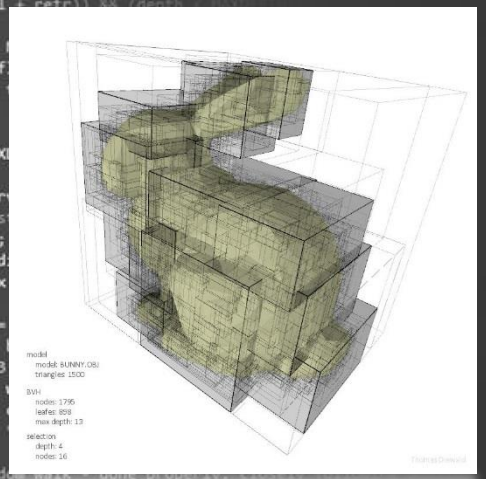


Today’s Agenda:

- Exam
- Digest
- Grand Recap
- Now What

Now What





Now What

/INFOMOV/