# /INFOMOV/

# Optimization & Vectorization

J. Bikker - Sep-Nov 2019 - Lecture 13: “Snippets”

# Welcome!

# Today’s Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments

# Self-modifying

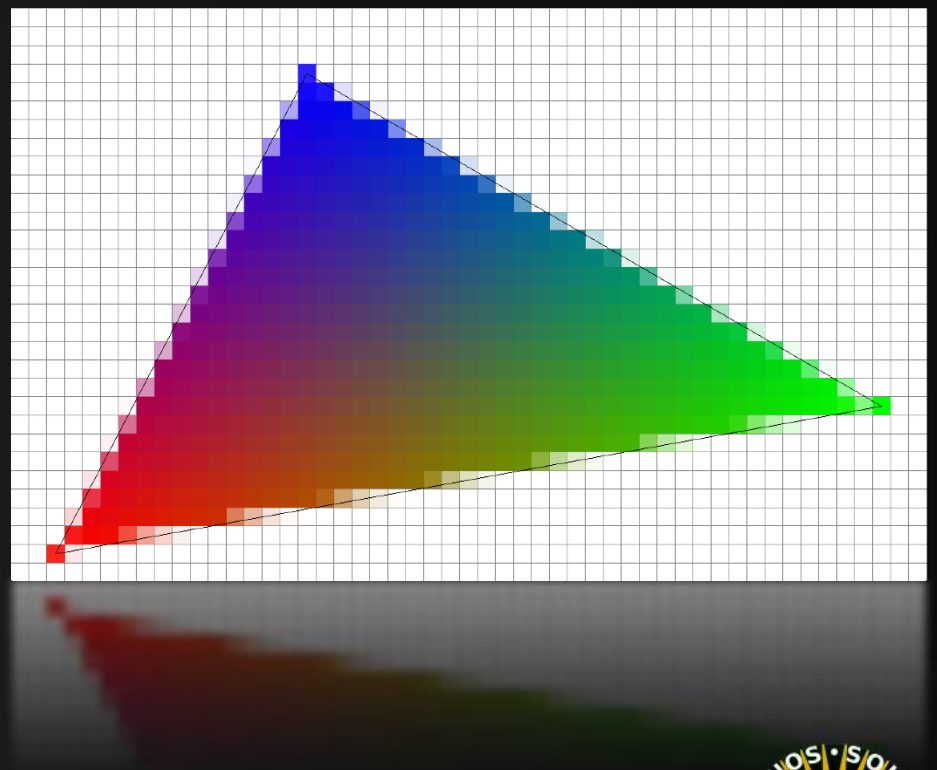
Fast Polygons on Limited Hardware

Typical span rendering code:

```
for( int i = 0; i < len; i++ )
{
    *a++ = texture[u,v];
    u += du;
    v += dv;
}
```

How do we make this faster?
Every cycle counts…

- ~~Loop unrolling~~
- Two pixels at a time
- …

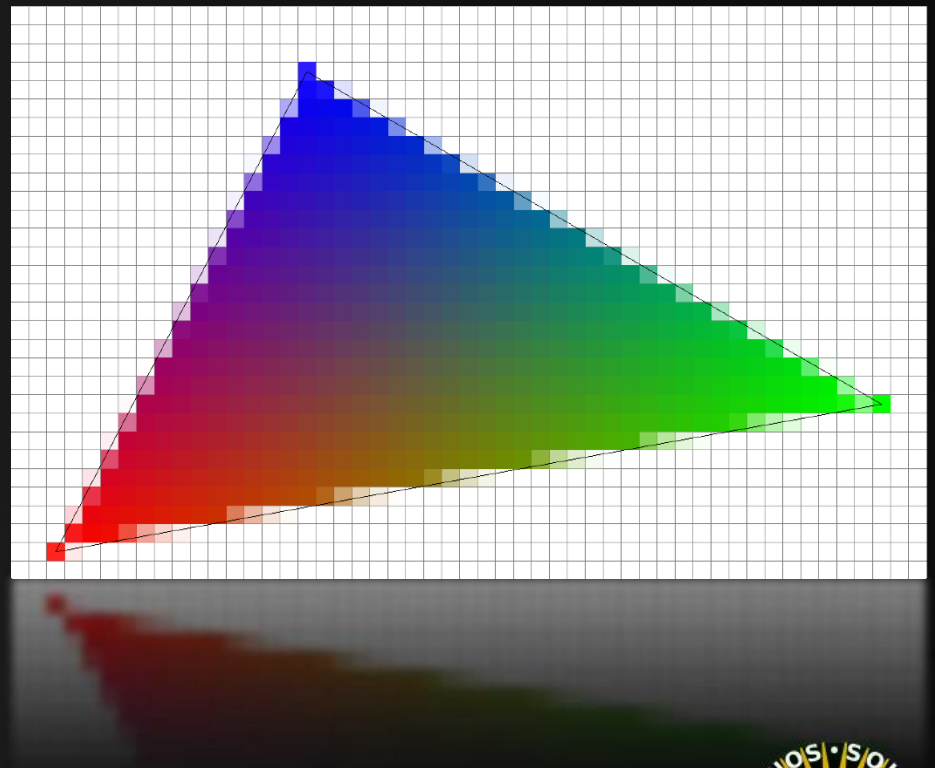


# Self-modifying

Fast Polygons on Limited Hardware

How about…

```
switch (len)
{
 case 8: *a++ = tex[u,v]; u+=du; v+=dv;
 case 7: *a++ = tex[u,v]; u+=du; v+=dv;
 case 6: *a++ = tex[u,v]; u+=du; v+=dv;
 case 5: *a++ = tex[u,v]; u+=du; v+=dv;
 case 4: *a++ = tex[u,v]; u+=du; v+=dv;
 case 3: *a++ = tex[u,v]; u+=du; v+=dv;
 case 2: *a++ = tex[u,v]; u+=du; v+=dv;
 case 1: *a++ = tex[u,v]; u+=du; v+=dv;
}
```



# Self-modifying

Fast Polygons on Limited Hardware

What if a massive unroll isn’t an option, but we have only 4 registers?

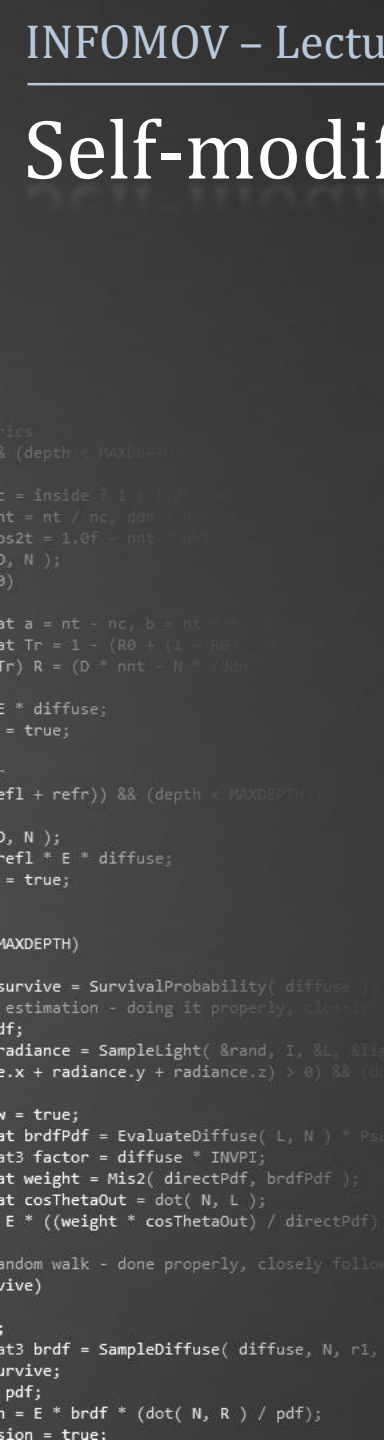
```
for( int i = 0; i < len; i++ )
{
   *a++ = texture[u,v];
   u += du, v += dv;
}
```

Registers: { i, a, u, v, du, dv, len }.

Idea: just before entering the loop,

- replace ‘len’ by the correct constant *in the code*;
- replace du and dv by the correct constant.

Our code is now *self-modifying*.



# Self-modifying

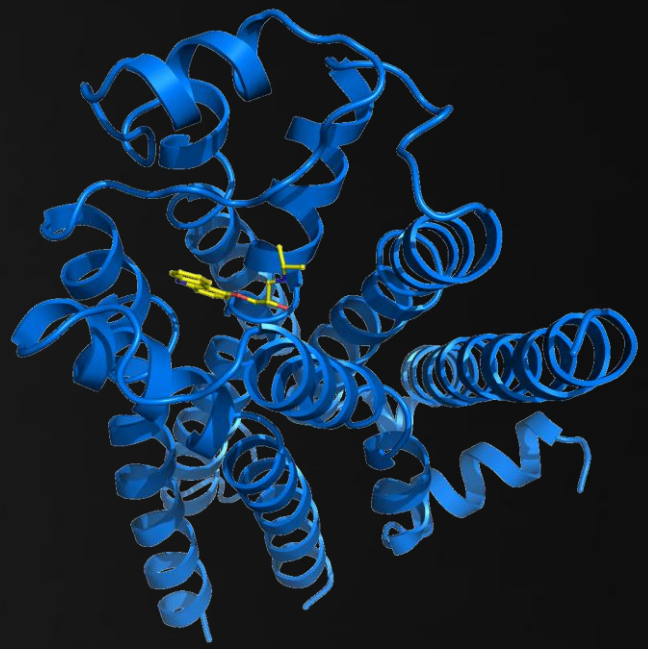
Self-modifying Code

Good reasons for **not** writing SMC:

- the CPU pipeline (mind every potential (future) target)
- L1 instruction cache (handles reads only)
- code readability

Good reasons for writing SMC:

- code readability
- genetic code optimization



# Self-modifying

Hardware Evolution*

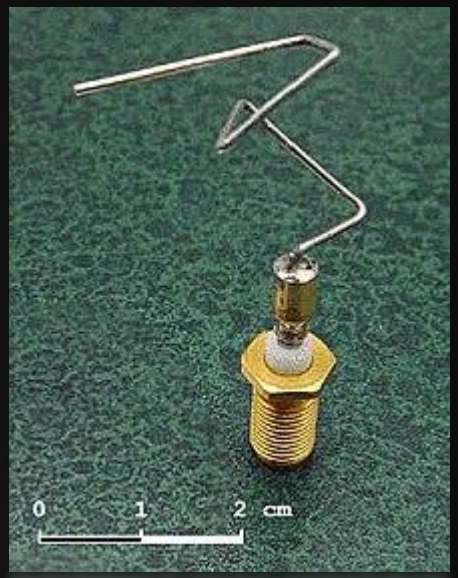
Experiment:

- take 100 FPGA’s, load them with random ‘programs’, max 100 logic gates
- test each chip’s ability to differentiate between two audio tones
- use the best candidates to produce the next generation.

Outcome (generation 4000): one chip capable of the intended task.

Observations:

1. The chip used only 37 logic gates, of which 5 disconnected from the rest.
2. The 5 disconnected gates where vital to the function of the chip.
3. The program could not be transferred to another chip.



NASA’s evolved antenna**

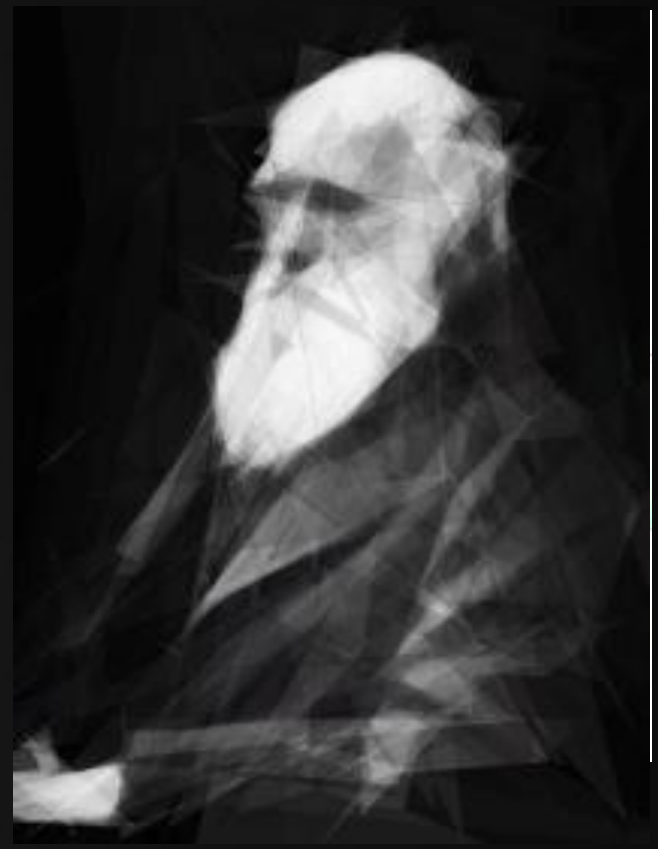
*: On the Origin of Circuits, Alan Bellows, 2007, https://www.damninteresting.com/on-the-origin-of-circuits
**: Evolved antenna, Wikipedia.

# Self-modifying

Compiler Flags*

Experiment:

“…we propose a genetic algorithm to determine the combination of flags, that could be used, to generate efficient executable in terms of time. The input population to the genetic algorithm is the set of compiler flags that can be used to compile a program and the best chromosome corresponding to the best combination of flags is derived over generations, based on the time taken to compile and execute, as the fitness function.”



*: Compiler Optimization: A Genetic Algorithm Approach, P. A. Ballal et al., 2015.

# Self-modifying

## Compiler Flags*

# Today’s Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments

# Multi-threading

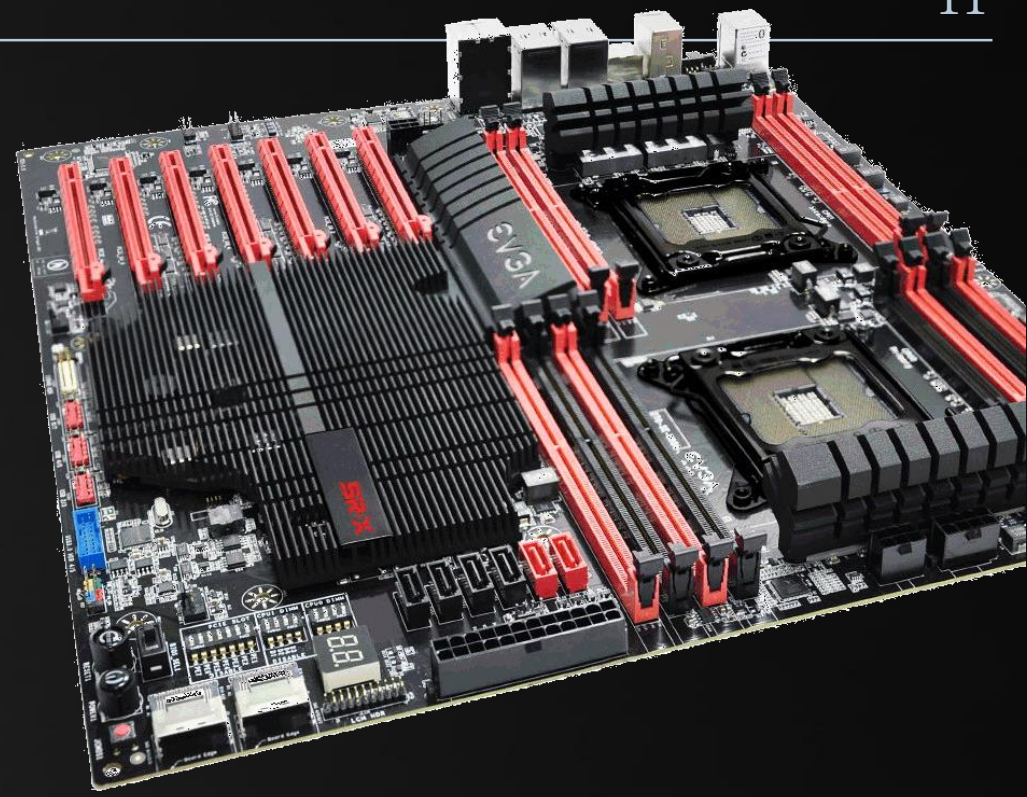
A Brief History of Many Cores

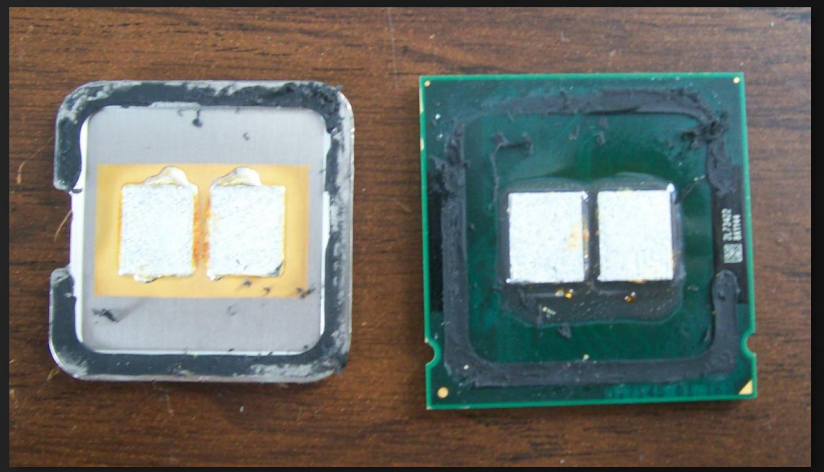
Once upon a time...

Then, in 2005: Intel’s Core 2 Duo *(April 22).*
*(Also 2005: AMD Athlon 64 X2. April 21.)*

*2007: Intel Core 2 Quad*

*2010: AMD Phenom II X6*





# Multi-threading

A Brief History of Many Cores

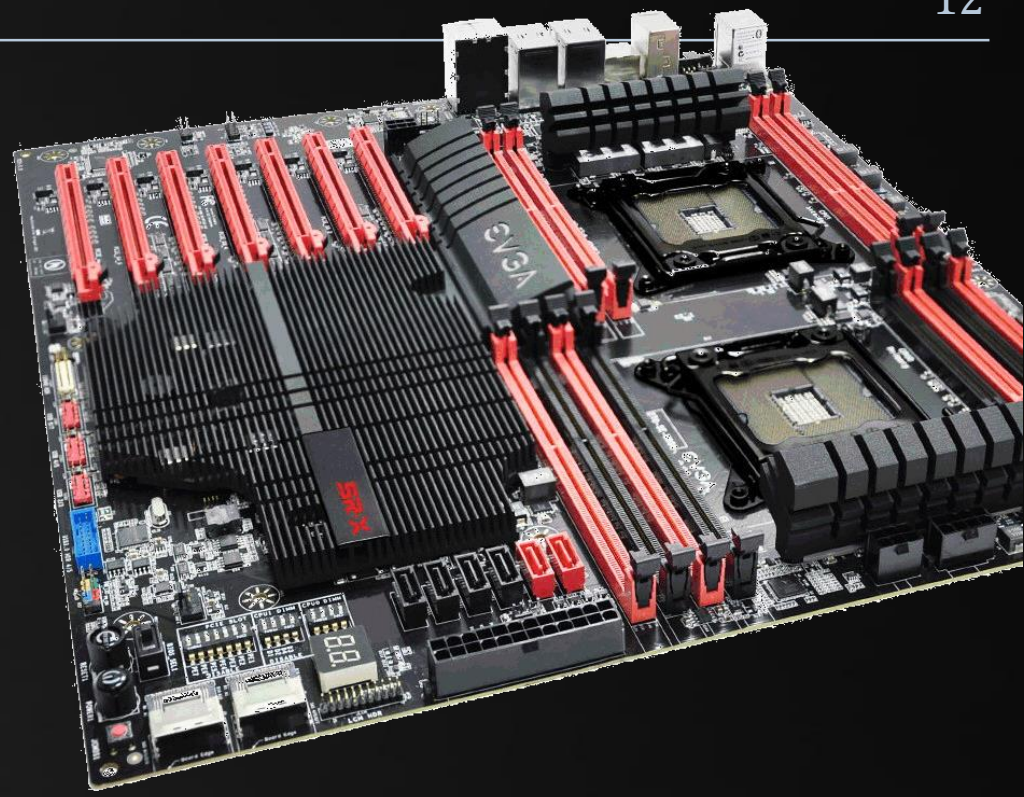
Once upon a time...

Then, in 2005: Intel’s Core 2 Duo *(April 22).*
*(Also 2005: AMD Athlon 64 X2. April 21.)*

*2007: Intel Core 2 Quad*

*2010: AMD Phenom II X6*

*Today...*






# Multi-threading

A Brief History of Many Cores

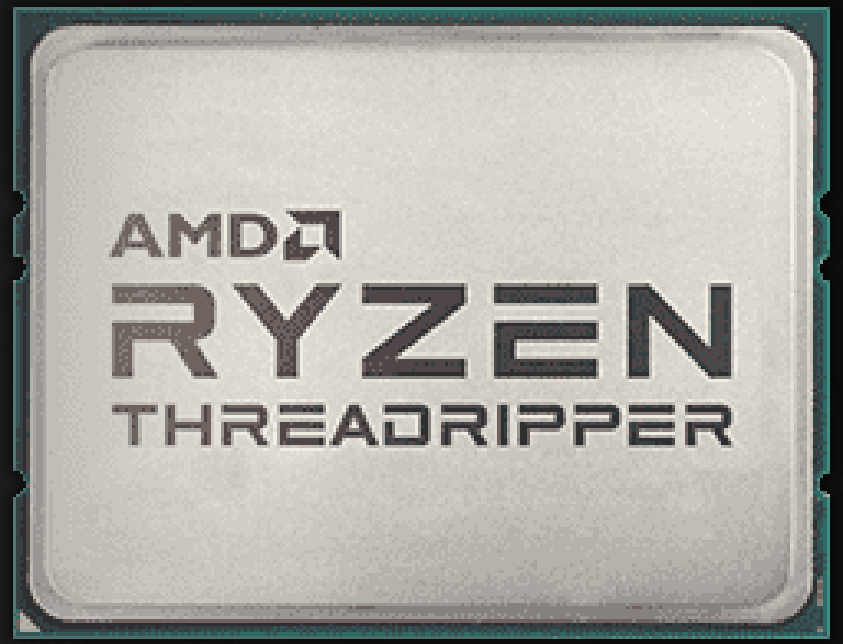
Once upon a time...

Then, in 2005: Intel’s Core 2 Duo *(April 22).*
*(Also 2005: AMD Athlon 64 X2. April 21.)*

*2007: Intel Core 2 Quad*

*2010: AMD Phenom II X6*

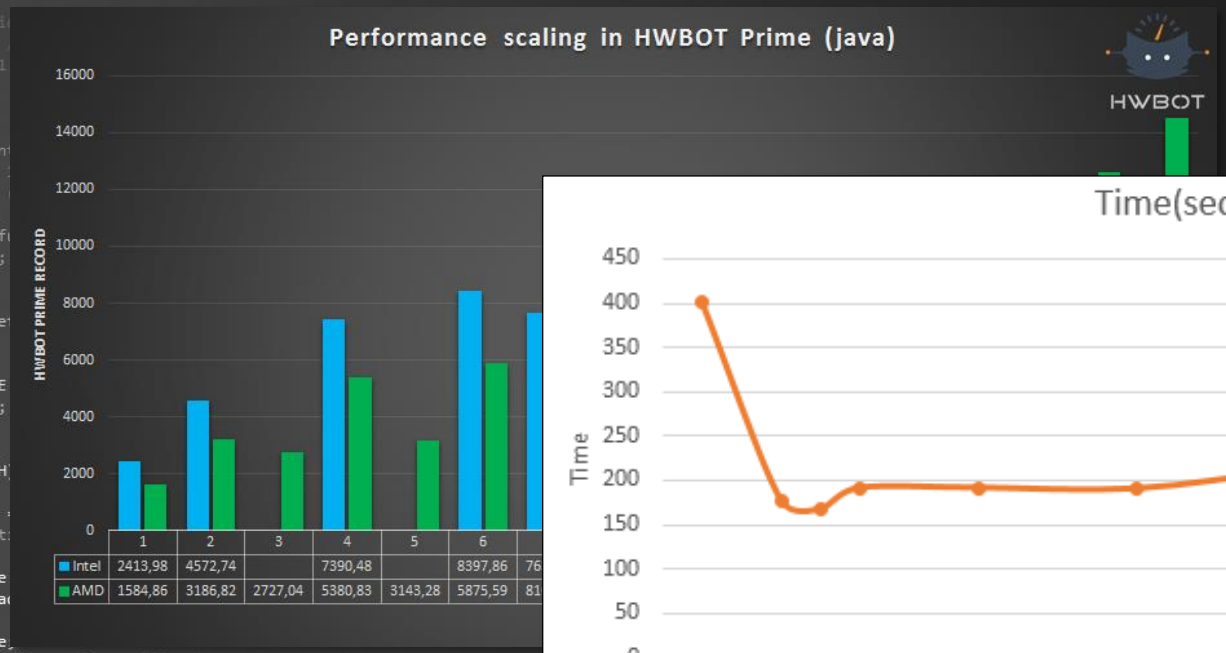
2017: Threadripper 1950X (16 cores, 32 threads)
2018: Threadripper 2950X
2019: Epyc 7742, 64 cores, 128 threads ($6,950)

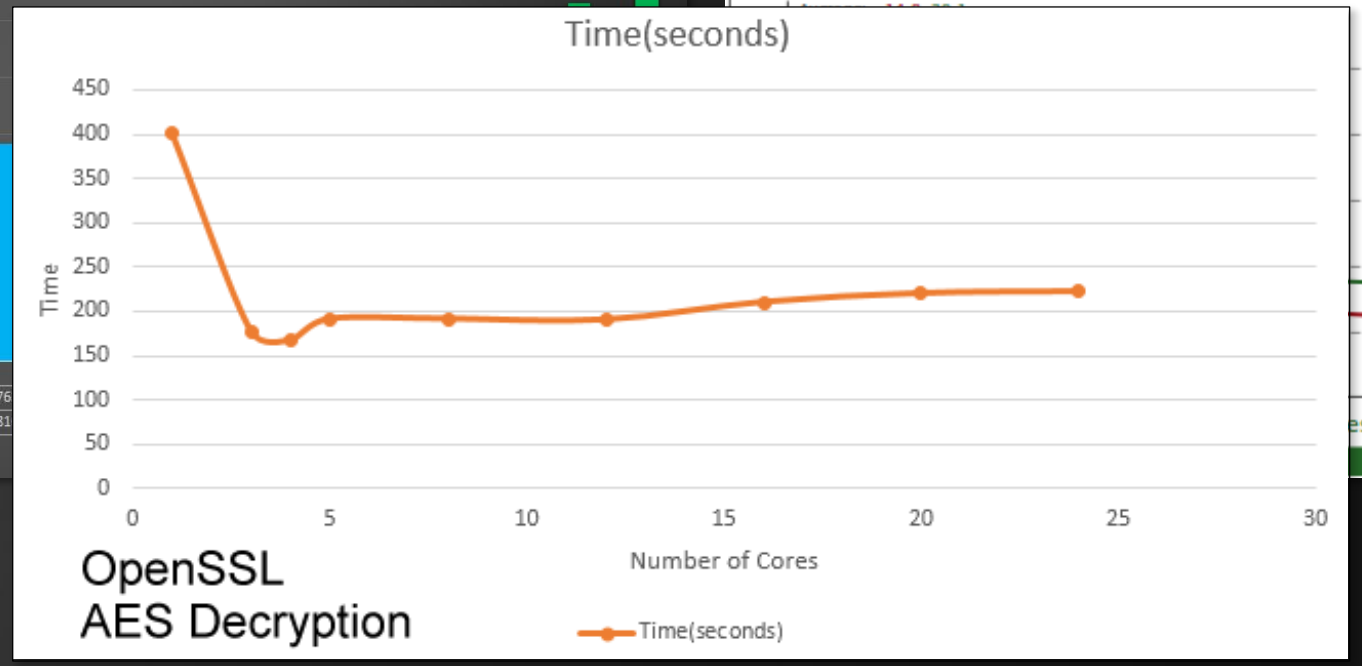


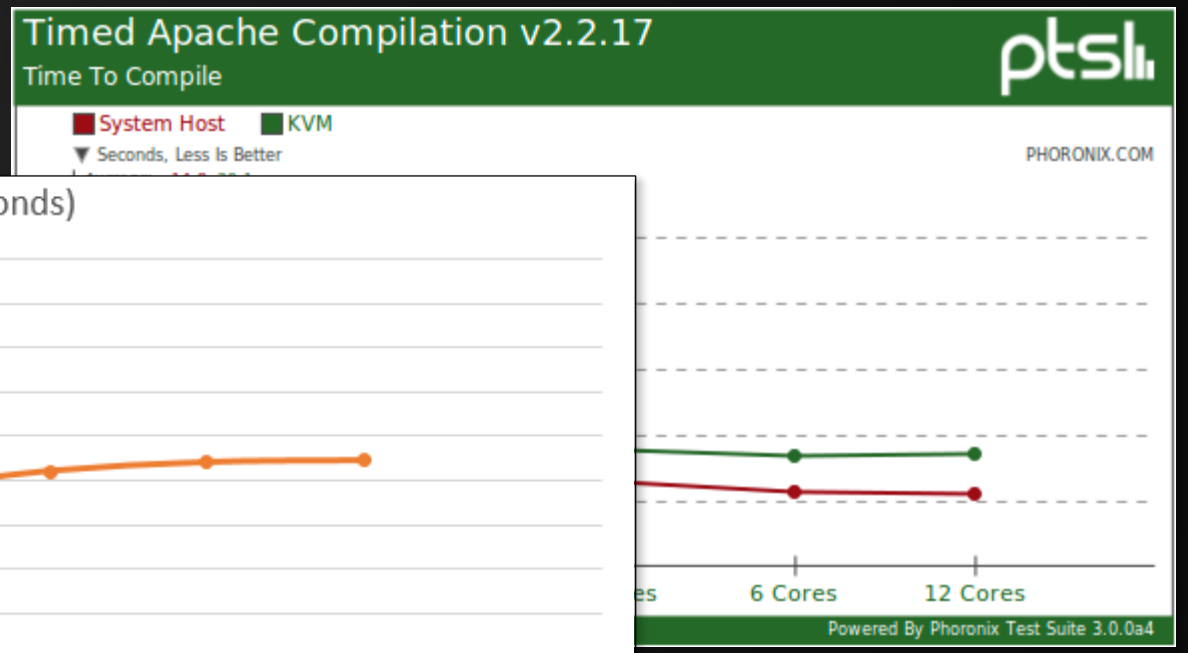

# Multi-threading

## Threads / Scalability

…



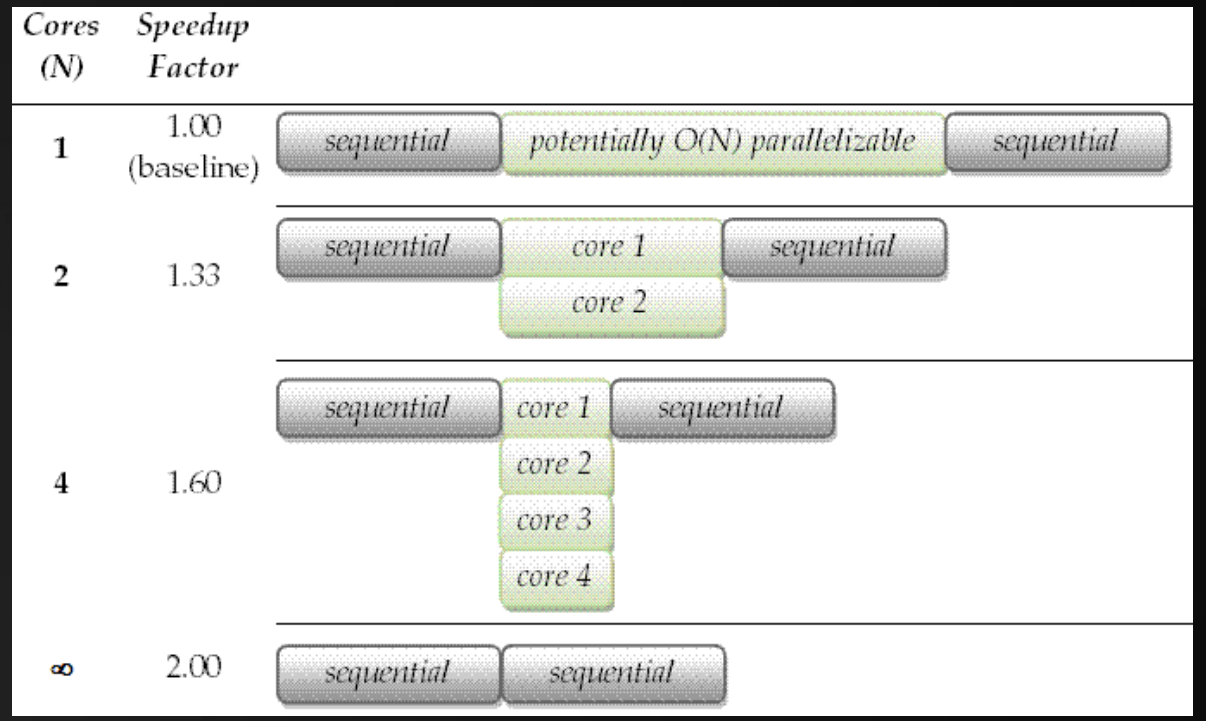



# Multi-threading

## Optimizing for Multiple Cores

What we did before:

1. Profile.
2. Understand the hardware.
3. Trust No One.




Goal:

- ~~It’s fast enough when it scales linearly with the number of cores.~~
- It’s fast enough when the parallelizable code scales linearly with the number of cores.
- It’s fast enough if there is no sequential code.

# Multi-threading

## Hardware Review

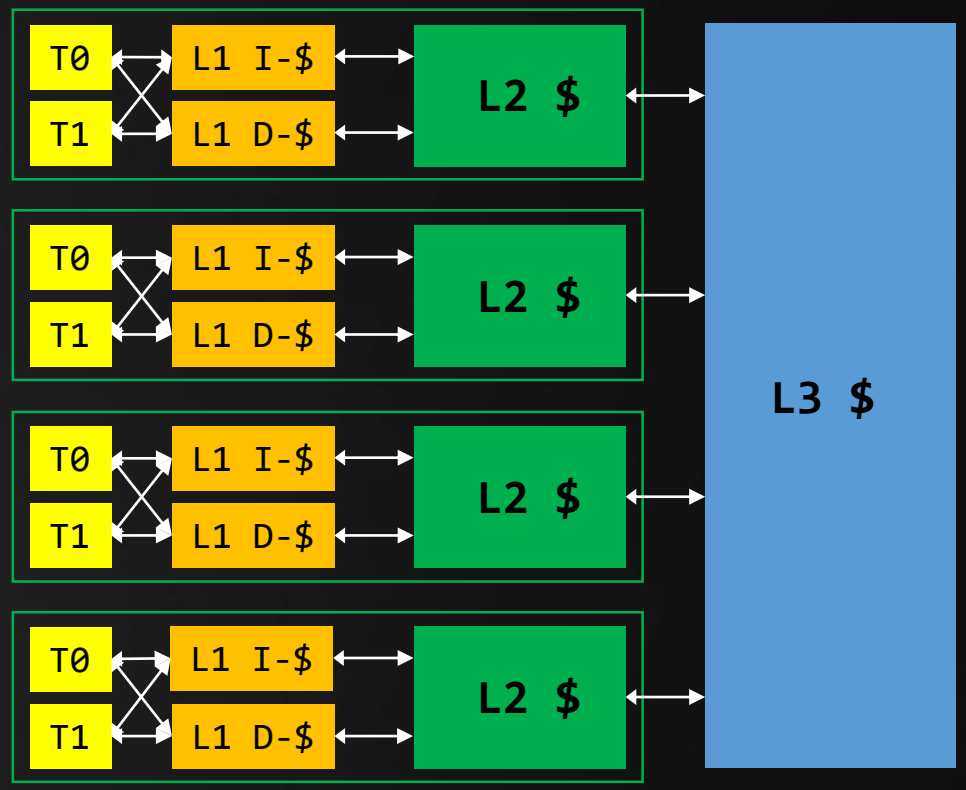
We have:

- Four physical cores
- Each running two threads
- L1 cache: 32Kb, 4 cycles latency
- L2 cache: 256Kb, 10 cycles latency
- A large shared L3 cache.

Observation:

*If our code solely requires data from L1 and L2, this processor should do work split over four threads exactly four times faster.*

*(Is that true? Any conditions?)*




- Work must stay on core
- No I/O, sleep
- …

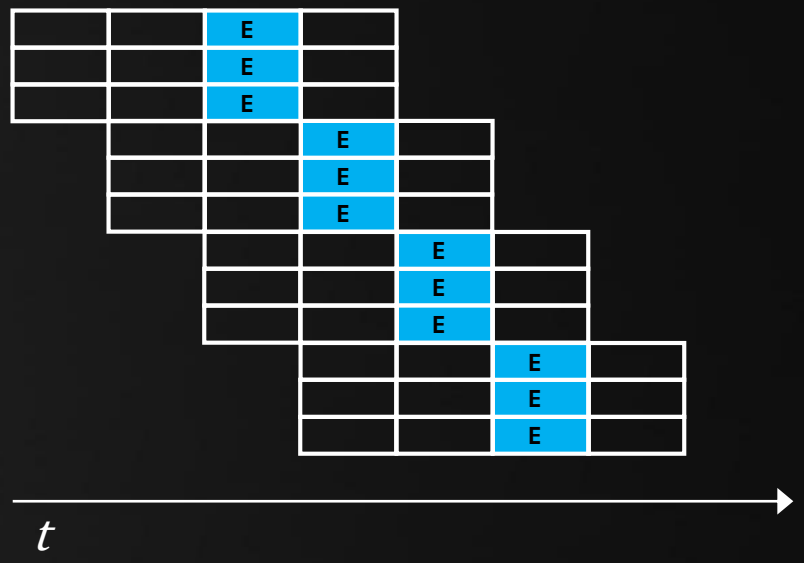
# Multi-threading

Simultaneous Multi-Threading (SMT)

(Also known as hyperthreading)

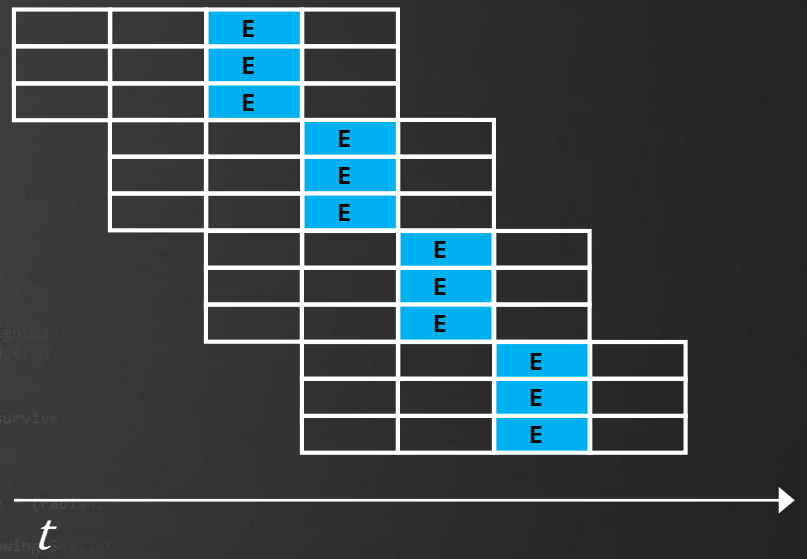
Pipelines grow wider and deeper:

- Wider: to execute multiple instructions in parallel in a single cycle.
- Deeper: to reduce the complexity of each pipeline stage, which allows for a higher frequency.



# Multi-threading

## Superscalar Pipeline




```
fldz
xor ecx, ecx
fld dword ptr [4520h]
mov edx, 28929227h
fld dword ptr [452Ch]
push esi
mov esi, 0C350h
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, 0Eh
dec esi
jne tobetimed+1Fh
```

# Multi-threading

Superscalar Pipeline

Nehalem (i7): *six* wide.

- Three memory operations
- Three calculations (float, int, vector)


```
fldz
xor ecx, ecx
fld dword ptr [4520h]
mov edx, 28929227h
fld dword ptr [452Ch]
push esi
mov esi, [0C350h]
add ecx, edx
mov eax, [91D2h]
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, 0Eh
dec esi
jne tobetimed+1Fh
```

# Multi-threading

Simultaneous Multi-Threading (SMT)

(Also known as hyperthreading)

Pipelines grow wider and deeper:

- Wider, to execute multiple instructions in parallel in a single cycle.
- Deeper, to reduce the complexity of each pipeline stage, which allows for a higher frequency.

However, parallel instructions must be independent, otherwise we get bubbles.

Observation: *two threads provide twice as many independent instructions.*

***(Is that true? Any conditions?)***


- No dependencies between the threads
- …

# Multi-threading

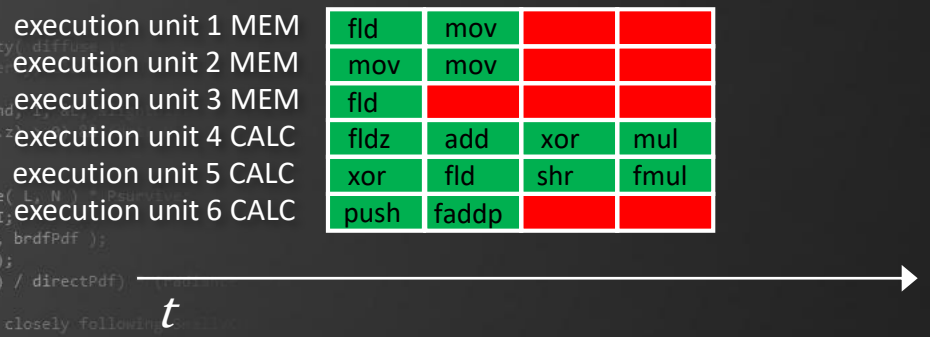
Simultaneous Multi-Threading (SMT)

Nehalem (i7) pipeline: *six* wide*.

- Three memory operations
- Three calculations (float, int, vector)

SMT: feeding the pipe from *two* threads.

All it really takes is an extra set of registers.




```
fldz
xor ecx, ecx
fld dword ptr [4520h]
mov edx, 28929227h
fld dword ptr [452Ch]
push esi
mov esi, 0C350h
add ecx, edx
mov eax, [91D2h]
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, 0Eh
dec esi
jne tobetimed+1Fh
```

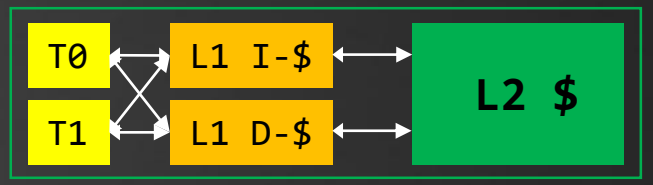
```
fld st(1)
faddp st(3), st
mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1),st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
fldz
xor ecx, ecx
fld dword ptr [4520h]
mov edx, 28929227h
fld dword ptr [452Ch]
push esi
mov esi, 0C350h
add ecx, edx
mov eax, [91D2h]
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
jne tobetimed+1Fh
```

*: Details: The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, Thomadakis, 2011.

# Multi-threading

Simultaneous Multi-Threading (SMT)

Hyperthreading does mean that now *two* threads are using the same L1 and L2 cache.




- For the average case, this will reduce data locality.
- If both threads use the same data, data locality remains the same.
- One thread can also be used to fetch data that the other thread will need *.

*: Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors, Luk, 2001.

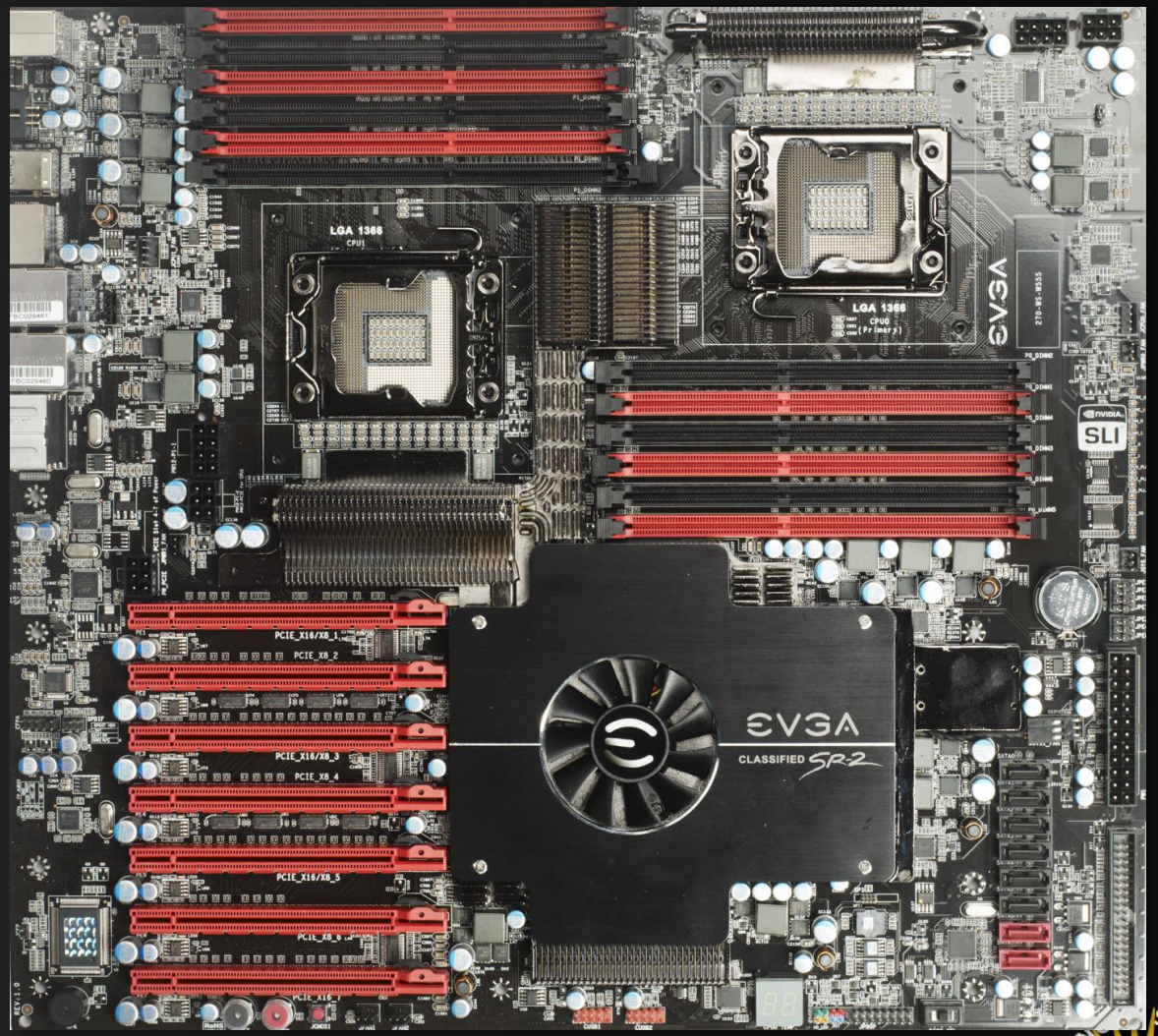
# Multi-threading

## Multiple Processors: NUMA

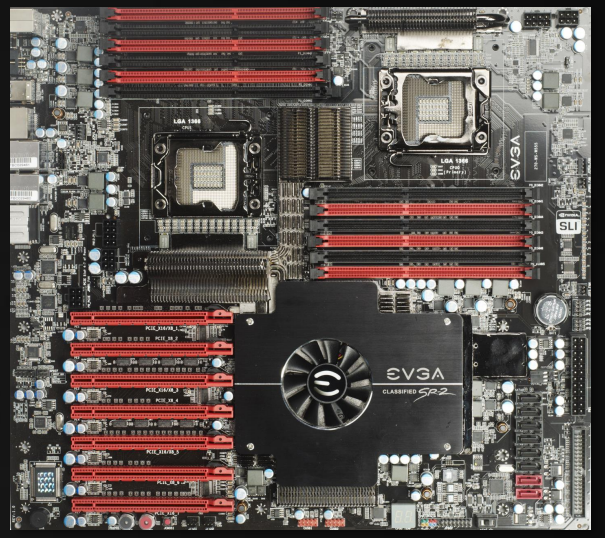
Two physical processors on a single mainboard:

- Each CPU has its own memory
- Each CPU can access the memory of the other CPU.

The penalty for accessing ‘foreign’ memory is ~50%.



# Multi-threading


Multiple Processors: NUMA

Do we care?

- Most boards host 1 CPU.
- A quadcore still talks to memory via a single interface.

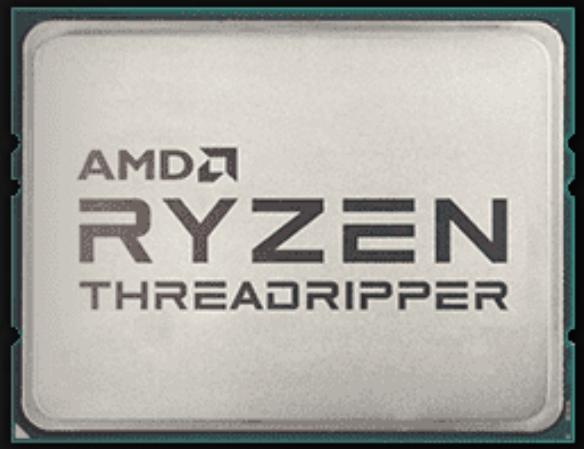
However:

Threadripper is a NUMA device.

Threadripper = 2x Zeppelin, with for each Zeppelin:

- L1, L2, L3 cache
- A link to memory

*This CPU behaves as two CPUs in a single socket.*

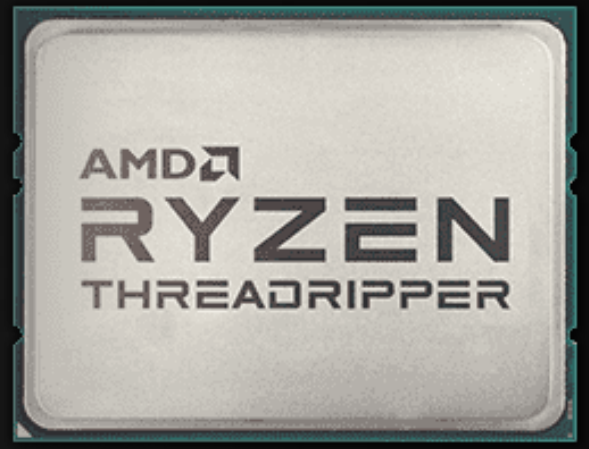


# Multi-threading

Multiple Processors: NUMA

Threadripper & Windows:

- Threadripper hides NUMA from the OS
- Most software is not NUMA-aware.




Details: https://www.extremetech.com/computing/283114-new-utility-can-double-amd-threadripper-2990wx-performance
https://blog.michael.kuron-germany.de/2018/09/amd-ryzen-threadripper-numa-architecture-cpu-affinity-and-htcondor

# Today’s Agenda:

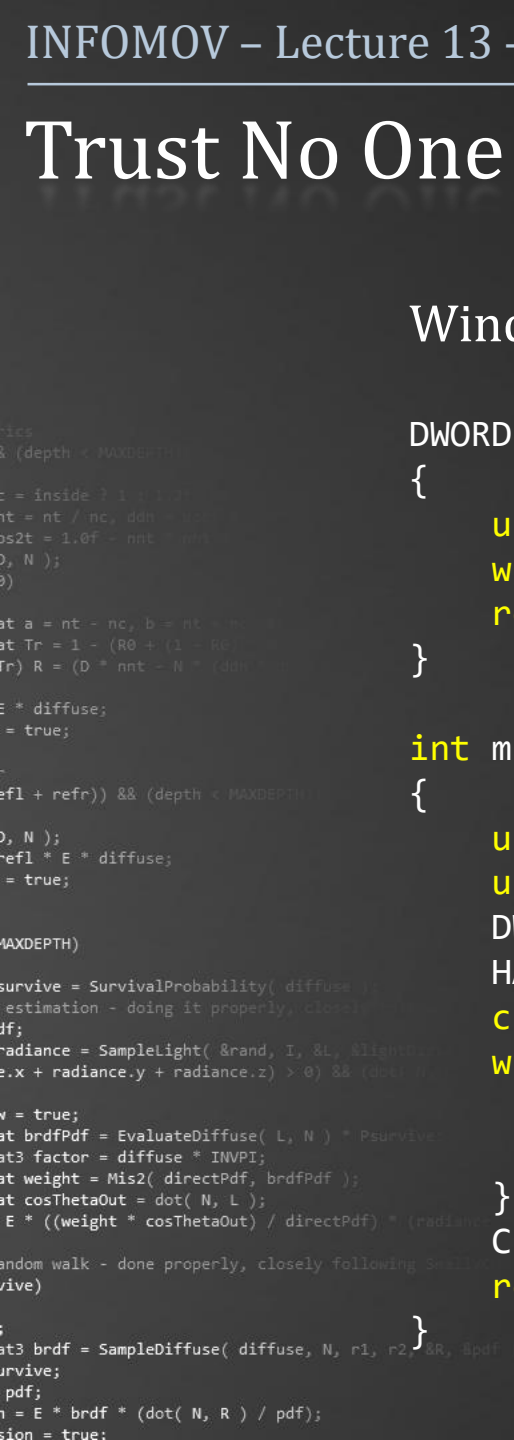
- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments

# Trust No One

## Windows

```
DWORD WINAPI myThread(LPVOID lpParameter)
{
    unsigned int& myCounter = *((unsigned int*)lpParameter);
    while(myCounter < 0xFFFFFFFF) ++myCounter;
    return 0;
}

int main(int argc, char* argv[])
{
    using namespace std;
    unsigned int myCounter = 0;
    DWORD myThreadID;
    HANDLE myHandle = CreateThread(0, 0, myThread, &myCounter;, 0, &myThreadID;);
    char myChar = ' ';
    while(myChar != 'q') {
        cout << myCounter << endl;
        myChar = getchar();
    }
    CloseHandle(myHandle);
    return 0;
}
```



# Trust No One

## Boost

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
    boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << '\n';
    }
}

int main()
{
    boost::thread t{thread};
    t.join();
}
```

# Trust No One

## OpenMP

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

float a[8], b[8];
#pragma omp simd
for( int n = 0; n < 8; ++n) a[n] += b[n];

struct node { node *left, *right; };
extern void process(node* );
void postorder_traverse(node* p)
{
    if (p->left)
        #pragma omp task
        postorder_traverse(p->left);
    if (p->right)
        #pragma omp task
        postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

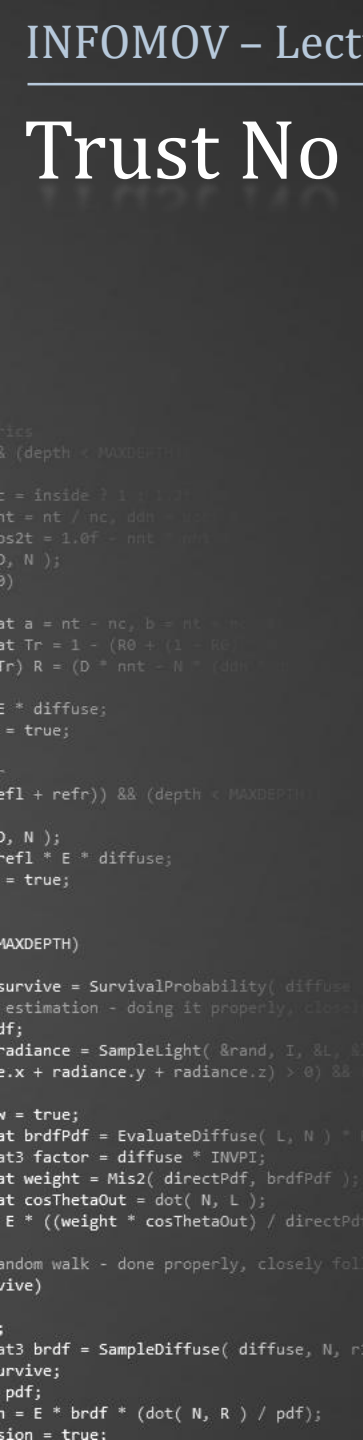
# Trust No One

## Intel TBB

```
#include "tbb/task_group.h"

using namespace tbb;

int Fib( int n )
{
    if (n<2)
    {
        return n;
    }
    else
    {
        int x, y;
        task_group g;
        g.run( [&]{x=Fib( n - 1 );} );  // spawn a task
        g.run( [&]{y=Fib( n - 2 );} );  // spawn another task
        g.wait();                       // wait for both tasks to complete
        return x + y;
    }
}
```



# Trust No One

## Considerations

When using external tools to manage your threads, ask yourself:

- What is the overhead of creating / destroying a thread?
- Do I even know when threads are created?
- Do I know on which cores threads execute?

*What if… we handled everything ourselves?*

# Trust No One


tasks:

- Worker threads never die
- Worker threads are pinned to a core
- Tasks are claimed by worker threads
- Execution of a task may depend on completion of other tasks
- Tasks can produce new tasks

# Trust No One

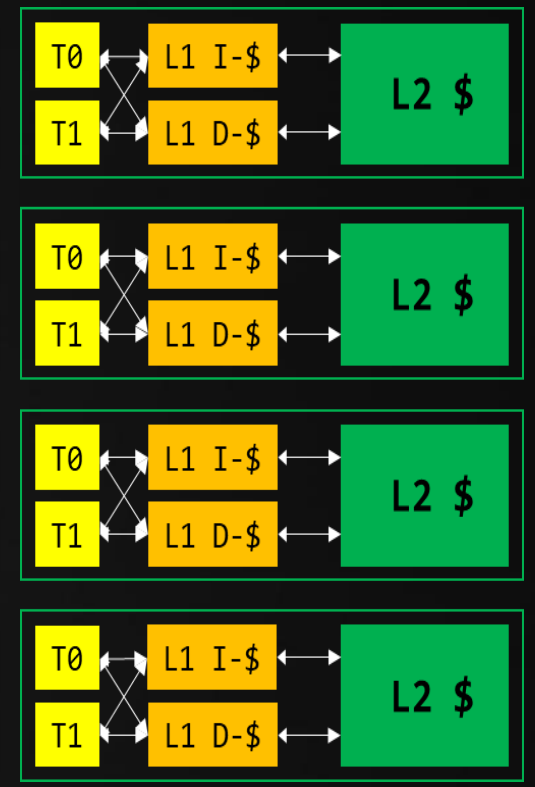
worker thread 0
worker thread 1
worker thread 2
worker thread 3
worker thread 4
worker thread 5
worker thread 6
worker thread 7

Fibers:

- Light-weight threads, with a complete state: registers (incl. program counter), stack
- Available in Windows, PS4, …
- Allows the task system to suspend a job, e.g. to wait for scheduled sub-tasks

Sub-tasks:

- Decrement a counter when done
- When counter reaches zero, linked task is resumed.




tasks:

- Worker threads never die
- Worker threads are pinned to a core
- Tasks are claimed by worker threads
- Execution of a task may depend on completion of other tasks
- Tasks can produce new tasks

# Trust No One

Fibers:

- “Cooperative multithreading”, no preemption

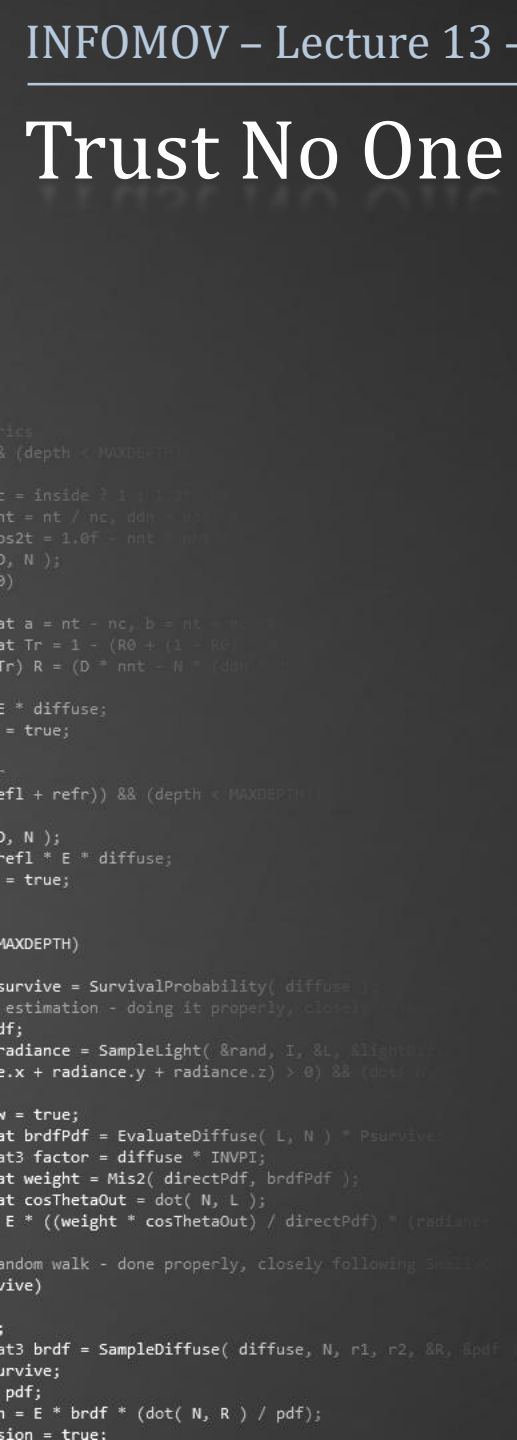
Fibers on Windows:

https://docs.microsoft.com/en-us/windows/win32/procthread/fibers

ConvertThreadToFiber
CreateFiber
SwitchToFiber

Cross-platform fibers:

https://github.com/JarkkoPFC/fiber



# Rules of Engagement

## Multithreading & Performance

- SMT / Hyperthreading: sharing L1 & L2 cache
  - Problems similar to simply having more threads
  - However, without the extra threads we don’t benefit from SMT
  - Mitigate: have the threads work on the same data
- Multiple cores
  - Threads may travel from one core to the next (mind the caches)
  - Must share bandwidth
  - Mind false sharing
- NUMA
  - Thread assignment now depends on what memory is used
  - No longer a theoretical issue
- Libraries
  - Generally favor ease of use over performance

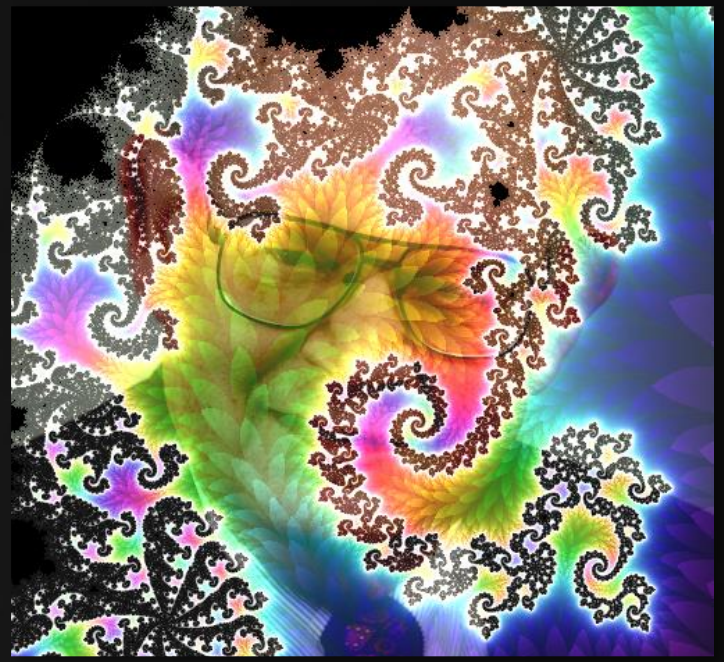
# Today’s Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments

# Experiments

## Trust No One

How fast does OpenMP make an ‘embarrassingly parallel’ application?



```
void Game::Tick( float deltaTime )
{
    // draw one line of pixels
    static int xtiles = SCRWIDTH / TILESIZE, ytiles = SCRHEIGHT / TILESIZE;
    static int tileCount = xtiles * ytiles;
    for( int i = 0; i < tileCount; i++ )            // #pragma omp parallel for
    {
        int tx = i % xtiles;
        int ty = i / xtiles;
        drawtile( screen, tx * TILESIZE, ty * TILESIZE );
    }
}
```

# Experiments

Trust No One

How fast does OpenMP make an ‘embarrassingly parallel’ application?

*Can we do better?*


```
void Game::Tick( float deltaTime )
{
   // draw one line of pixels
   static int xtiles = SCRWIDTH / TILESIZE, ytiles = SCRHEIGHT / TILESIZE;
   static int tileCount = xtiles * ytiles;
   for( int i = 0; i < tileCount; i++ )            // #pragma omp parallel for
   {
      int tx = i % xtiles;
      int ty = i / xtiles;
      drawtile( screen, tx * TILESIZE, ty * TILESIZE );
   }
}
```

# Experiments

## Worker Threads


```
static DWORD threadId[THREADCOUNT];
static int params[THREADCOUNT];
static HANDLE worker[THREADCOUNT];
// spawn worker threads
for( int i = 0; i < 4; i++ )
{
   params[i] = i;
   worker[i] = CreateThread( NULL, 0, workerthread, &params[i], 0, &threadId[i] );
}
```

# Experiments

## Worker Threads

```
HANDLE goSignal[4], doneSignal[4];

volatile LONG remaining = 0;

unsigned long __stdcall workerthread( LPVOID param )
{
   int threadId = *(int*)param;
   while (1)
   {
      WaitForSingleObject( goSignal[threadId], INFINITE );
      while (remaining > 0)
      {
         int task = (int)InterlockedDecrement( &remaining ) - 1;
         if (task >= 0)
         {
            int tx = task % xtiles, ty = task / xtiles;
            drawtile( theScreen, tx * TILESIZE, ty * TILESIZE );
         }
      }
      SetEvent( doneSignal[threadId] );
   }
}
```

# Experiments

## Worker Threads

```
remaining = tileCount;
for( int i = 0; i < 4; i++ ) SetEvent( goSignal[i] );
WaitForMultipleObjects( THREADCOUNT, doneSignal, true, INFINITE );
```


# Today’s Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments

# /INFOMOV/

## END of “Snippets”

next lecture: “Exam Practice”