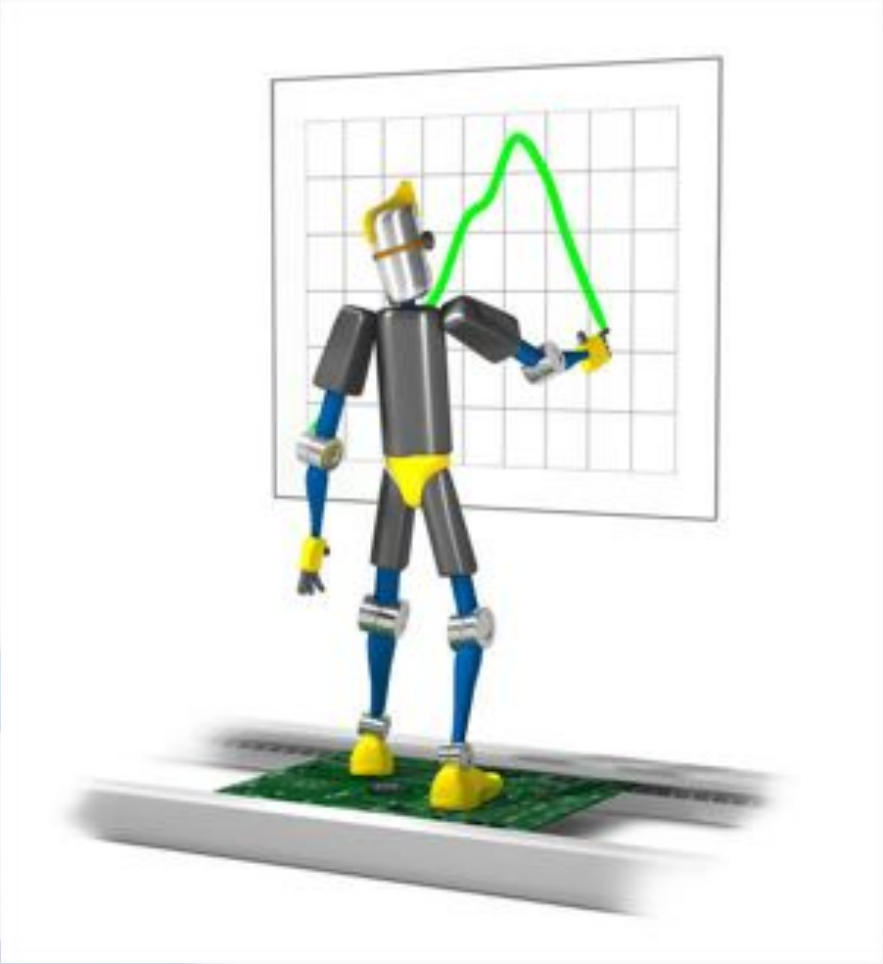
Profiling and Optimization In Medical Imaging

Presented By: Muhammad Adel

Lead Software Engineer

Outline

- Why 3D imaging
- Profiling
  - CPU
  - GPU
- Optimization
  - Compiler
  - Developer
- Final Words



SECTION 1: PROFILING

3D IMAGINING AND THE NEED FOR OPTIMIZATION

Features of 3D Imaging

- Large data sizes
- Computationally intensive operations
  - Image processing
  - Computational geometry
- Rendering
  - We need 40 FPS or more

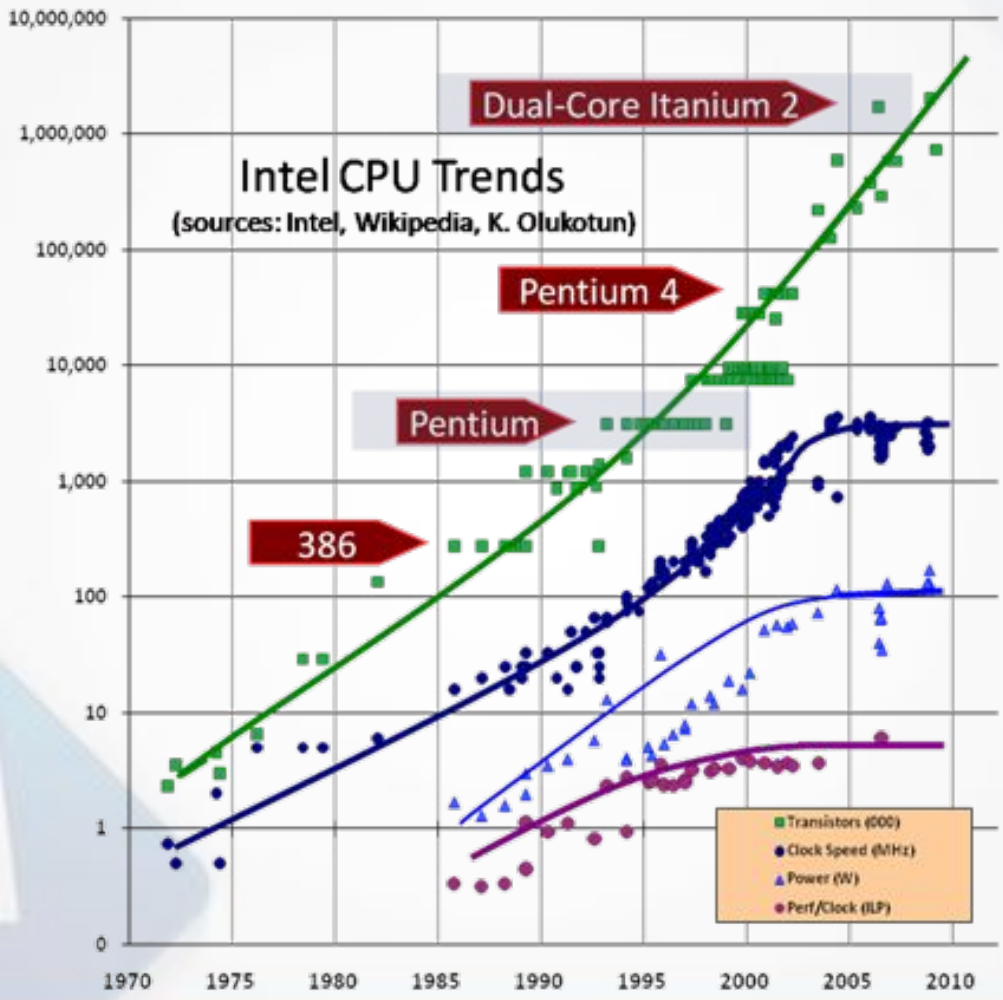
CPU side processing Operations

- Load code from IO to memory.
- Load data from IO to memory.
- Load code from Memory to cache.
- Load data from memory to cache.

CPU side processing Operations

- Processor speed increase at a higher rate than memory and cache access speed.
- Cache hit and miss.
- Branch prediction.

MOORE's LAW


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

IF WE CANNOT INCREASE SPEED LETS INCREASE THE NUMBER OF CORES

Multi core = Multi threading

- You have to write multithreaded code to get use of the multiple cores.
  - The free lunch is over.
- int x=0;
  //threaded part
  x++;
- What does this code do in detail?
  - Get x from memory.
  - Add 1
  - Return x to memory

Where is the problem?

- Problem with multithreading
  - Thread 1: get x from memory to my cache. X=0.
  - Thread 2: get x from memory to my chache. X = 0
  - Thread 1: add 1 to x. x=1.
  - Thread 2: add 1 to x. x = 1.
  - Thread 1: write x to memory. X=1.
  - Thread 2: write x to memory. X =1.
- The code was executed twice but x is still 1.
- What will happen if the two threads are not in sync?
  - One thread writing while the other is reading. Crash.
  - Bugs are not reproducible

Where is the problem?

- The code was executed twice but x is still 1.
- What will happen if the two threads are not in sync?
  - One thread writing while the other is reading. Crash.
- Some algorithms are not **THREAD SAFE**.

CPU PROFILING

What is profiling?

- Dynamic Program analysis
- Measure:
  - Memory used
  - Time of execution
  - CPU cycles consumed
  - And many other things
- In plain words, When the program is running slowly or consuming a large memory size, which part in the code is the reason?

Hands On

CALL GRAPH PROFILING

Hands On

PERFORMANCE COUNTERS PROFILING

GPU Profiling

- Less debugging and profiling capabilities than CPU
  - Data transfer between CPU and GPU
    - Locking
  - Parallel operations
  - Drivers

Hands on

GPU PROFILING

BREAK

SECTION 2: OPTIMIZATION

V Table

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

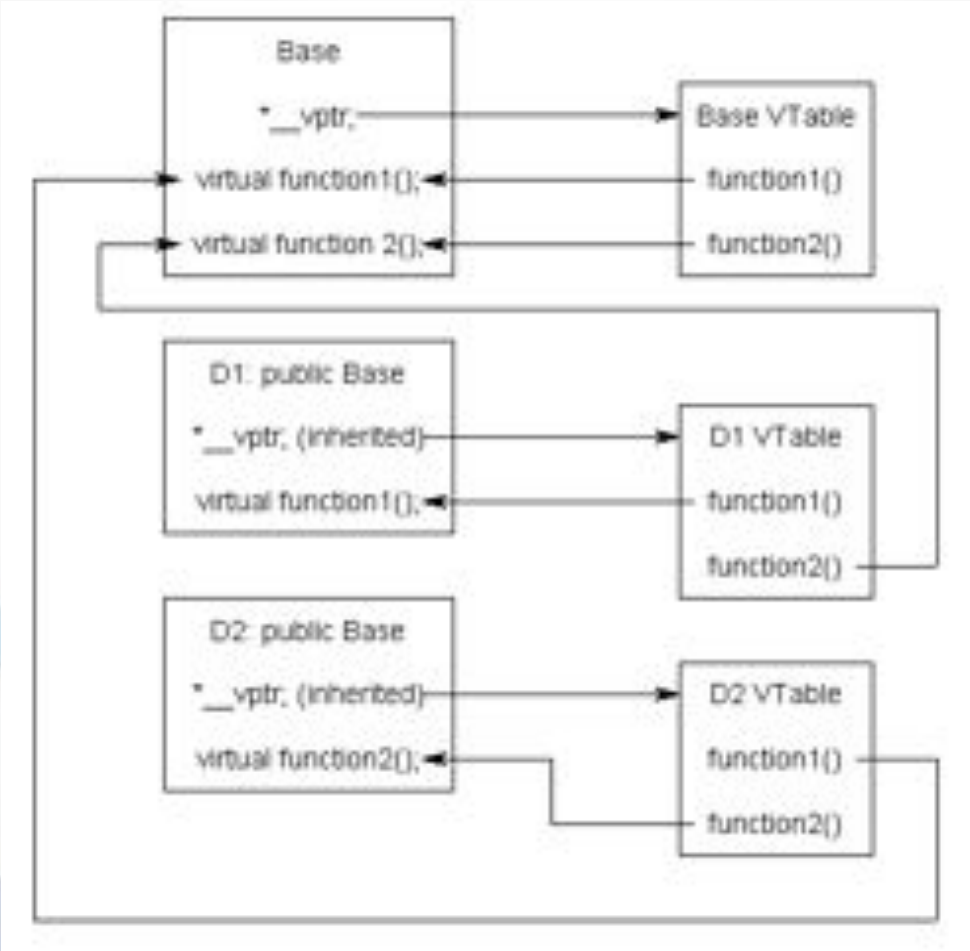
V Table

```
class Base
{
public:
static FunctionPointer *__vptr;
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   virtual void function1() {};
};

class D2: public Base
{
public:
   virtual void function2() {};
};
```

V Table


Base
*__vptr;
virtual function1();
virtual function 2();
Base VTable
function1()
function2()
D1: public Base
*__vptr; (inherited)
virtual function1();
D1 VTable
function1()
function2()
D2: public Base
*__vptr; (inherited)
virtual function2();
D2 VTable
function1()
function2()

Virtual function overhead

- Every call to a virtual function is preceded by a V Table lookup step.
- Prevents branch prediction.
- Increase cache miss.
- Same thing applies to function pointers

RULE 1

DON’T WRITE A VIRTUAL FUNCTION IF IT IS SHORT AND CALLED FREQUENTLY

Function Inlining

```
int multiply (int a, int b)
{
return a*b;
}
void main()
{
int x =5;
int y = 10;
int result = multiply(x,y);
cout<< result;
}
```

Function Inlining

```
inline int multiply (int a, int b)
{
return a*b;
}
void main()
{
int x =5;
int y = 10;
int result = multiply(x,y);
cout<< result;
}
```

Function Inlining

```
void main()
{
int x =5;
int y = 10;
int result = x*y;
cout<< result;
}
```

Function Inlining

- Saves a jump instruction
- Better performed on small function called frequently.
- According to the standards, inline C++ keyword is a hint for the compiler. It is not obliged to obey you.

RULE 2

INLINE SHORT FUNCTIONS THAT ARE CALLED FREQUENTLY

A good design is not necessarily efficient

- Object-orientedness is just a syntactic sugar on the top of procedural language, making the code more understandable to the human brain which seems to have evolved natural ability to deal with objects at the expense of dealing with entities such as flows, relations,etc.
- The ideology of the object-orientedness makes developers think in specific ways and design their algorithms and data structures accordingly. That in turn can have performance implications.

A good design is not necessarily efficient

- C:

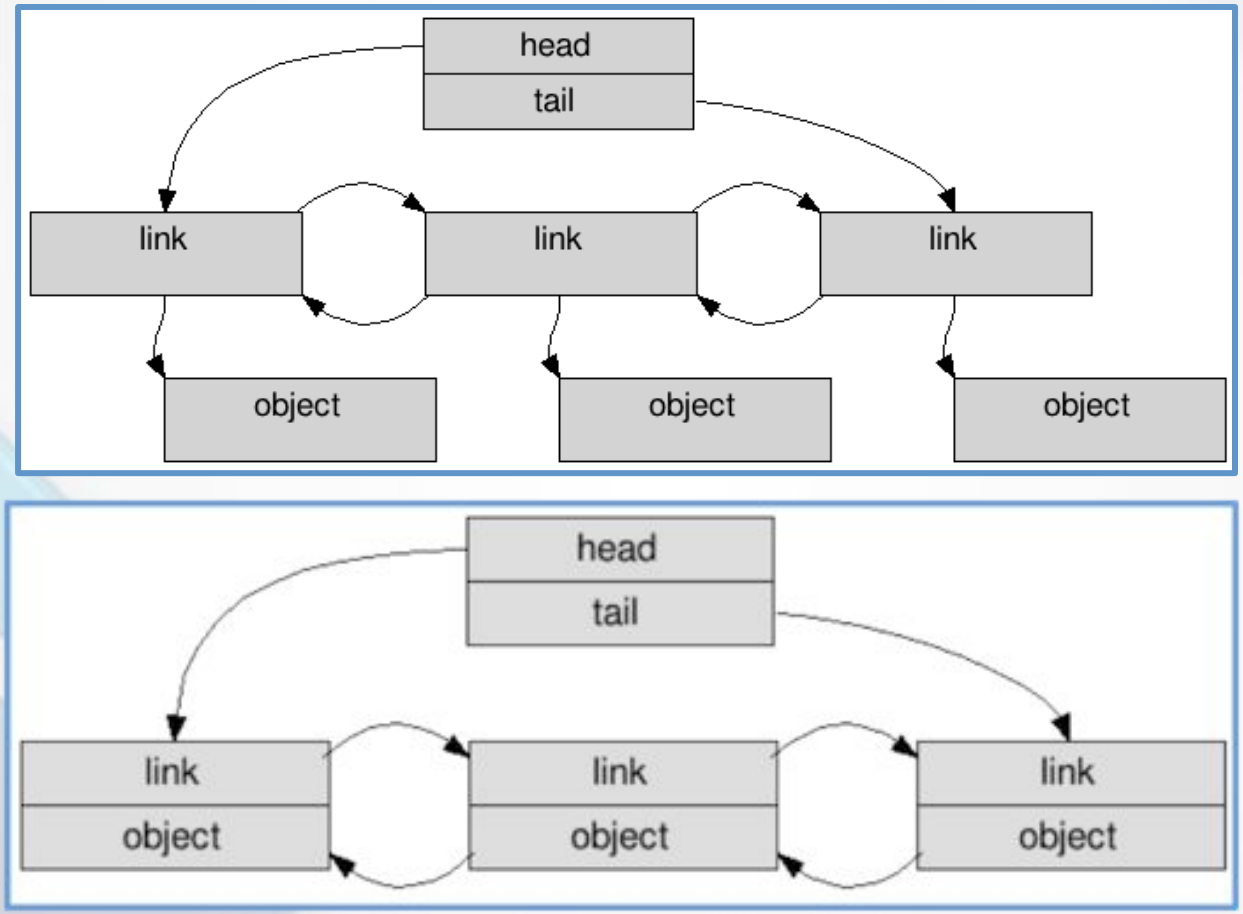
```
struct person
 {
struct person *prev;
 struct person *next;
 int age; int weight;
};
struct {
struct person *first;
struct person *last; } people;
```

A good design is not necessarily efficient

- C++:

```
class person
 {
int age; int weight;
 };
 std::list <person*> people;
```

A good design is not necessarily efficient



A good design is not necessarily efficient

- To use with **intrusive linked list** :
  struct person {
  TLink link; // The "intrusive" link field
  unsigned age; unsigned weight; };
- Boost has a whole namespace called Boost.Intrusive

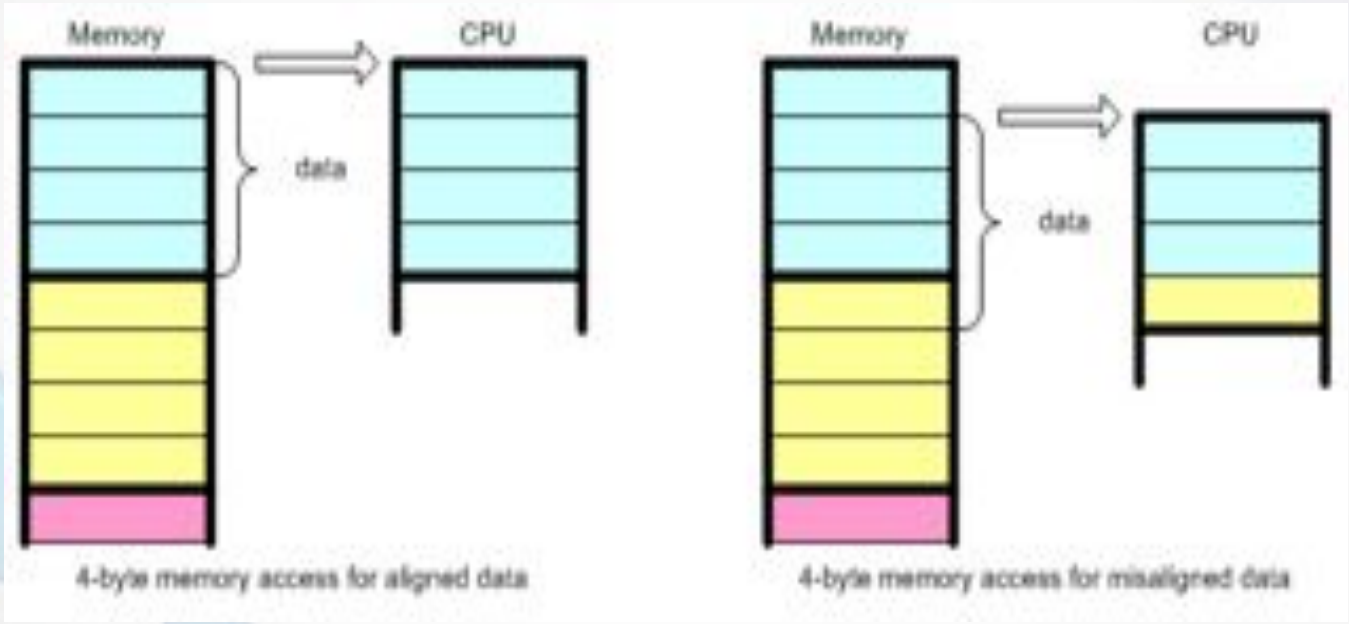
A good design is not necessarily efficient

- The second design is better from performance point of view because:
  - it is no longer necessary to allocate memory to link an item onto a list, nor deallocate memory when unlinking. **Lower memory usage and fewer allocation/deallocation.**
  - When traversing objects stored on an intrusive linked list, it only takes one pointer indirection to get to the object, compared to two pointer indirections for std::list. **Reduces cache misses**.

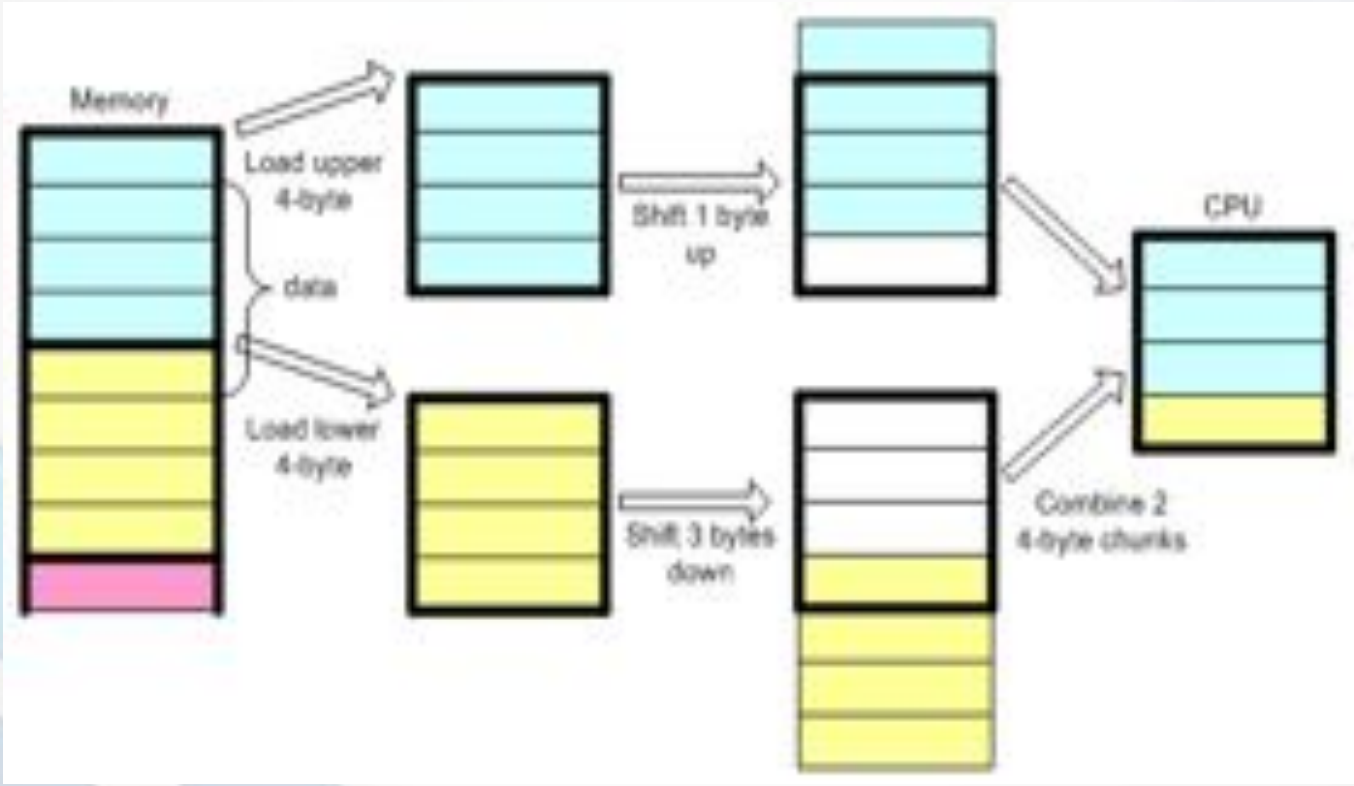
Rule 3

ALWAYS SUSPECT FREQUENTLY ACCESSED OBJECT ORIENTED DATA STRUCTURES

Data structures alignment



Data structures alignment



Data structure alignment

- Class myClass
  {
  float x,y,z,forPadding;
  }
- __declspec(align(32)) Class myClass
  {
  float x,y,z;
  }

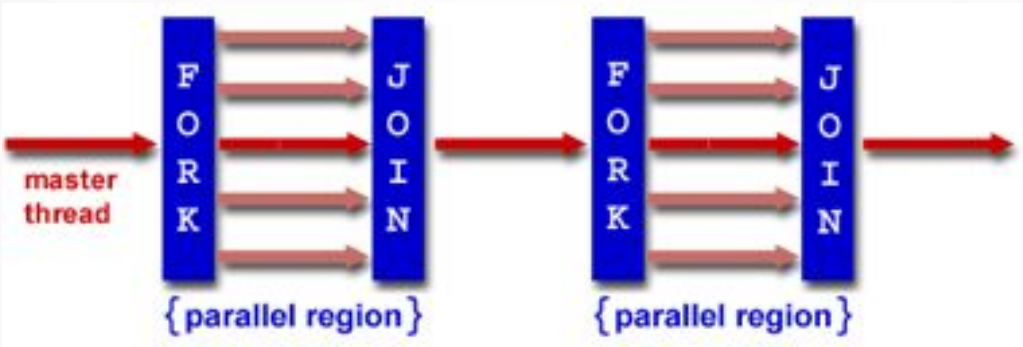
Rule 4

ALIGN SSE PROCESSED DATA STRUCTURES ON 16 BYTES MULTIPLES

Open MP

- #pragma omp parallel for
- for (int i=0;i<volumeSize;i++)
- {
- if (voxelValueInsideThresholdRange (i))
- m_VoxelsSelectionResults[i]=true;
- else
- m_VoxelsSelectionResults[i]=false;

Open MP


master
thread
FORK
JOIN
{parallel region}
FORK
JOIN
{parallel region}

OPEN MP

- Automaic thread handling
- Standard
  - *OpenMP Architecture Review Board: IBM, Microsoft, AMD, Intel, Oracle, TI, nVidia, Fujitsu, HP, NEC, etc.*
- We use it in 10 locations in our code.

Rule 5

USE OPEN MP FOR LOOP BASED THREAD SAFE ALGORITHMS

COMPILER OPTIMIZATIONS

Profile Guided Optimization

- Sample Code

```
void functionA(int input)
{
if (input >10)
functionB();
else
functionC();
```

- Which function to inline?

Profile Guided Optimization

- Create a special version of the exe that has **Probes**
  - **Path and value probes**
- Give it to the customer to use in real life scenarios.
- Get the collected data, give them to the compiler to create an optimized version

Problem

USAGE SCENARIOS DIFFER FROM ONE CUSTOMER TO ANOTHER

Just in time Compilation JIT

- Profile guided optimization on the fly
  - Requires balance between profiling and recompilation costs and the expected optimization benefit.
- Machine specific instructions

Other Automatic Optimizations

- Whole program optimization
- Garbage collector

Hands On

PUTTING IT ALL TOGETHER

Final Words

EARLY VS. LATE OPTIMIZATION

Thank you

Questions?

Symbyo
TECHNOLOGIES

Keep in Touch with us At:

/Symbyo

/Symbyo