How to save the world while programming?

7 ‘tips’ on sparing CPU cycles

Juanmi Huertas
R&D Software Engineer
Color and Imaging team
HP

Introduction

Seven topics to **know** to better use the CPU in your applications

But… why do we care about CPU?

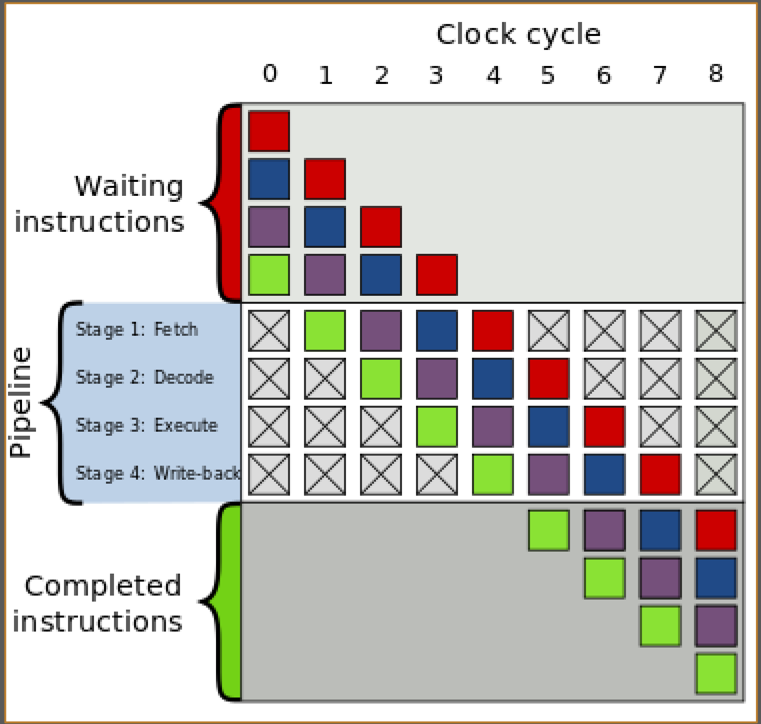
1) Know your hardware: CPU

“The only bad thing about theoretical computing is that there are no theoretical computers.”
Andy Thomason

CPU pipeline

Do not help the compiler!

But don’t sabotage it!



1) Know your hardware: CPU

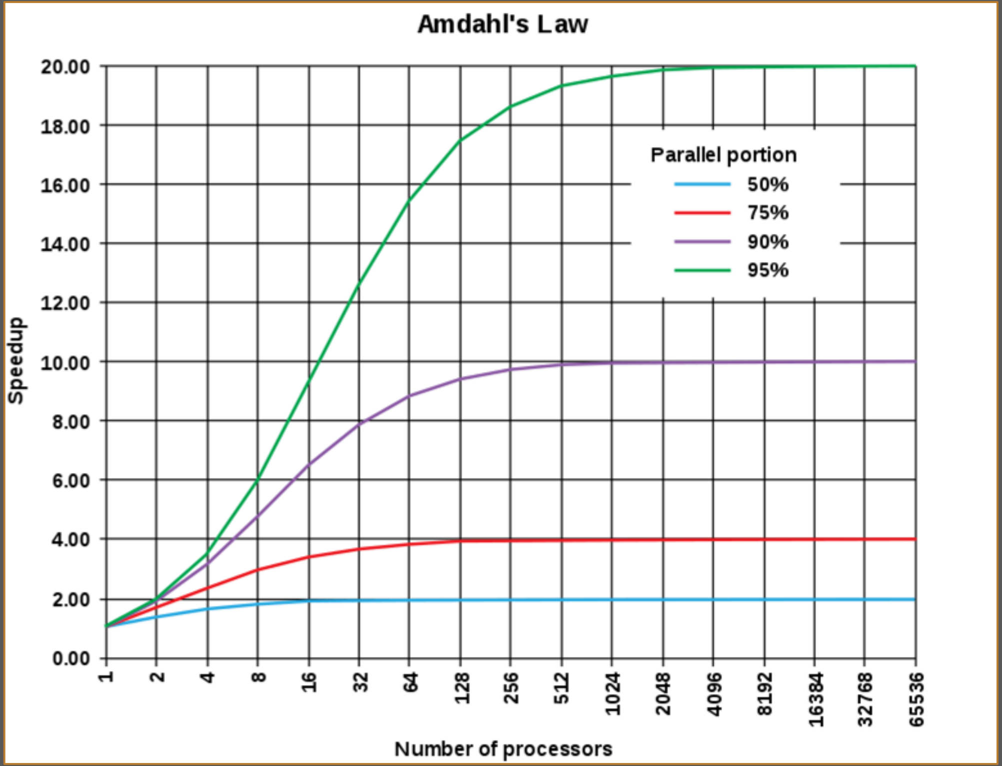
“The only bad thing about theoretical computing is that there are no theoretical computers.”
Andy Thomason

CPU pipeline

Do not help the compiler!

But don’t sabotage it!

Concurrency.



1) Know your hardware: CPU

“The only bad thing about theoretical computing is that there are no theoretical computers.”
Andy Thomason

CPU pipeline

Do not help the compiler!
But don’t sabotage it!

Concurrency?

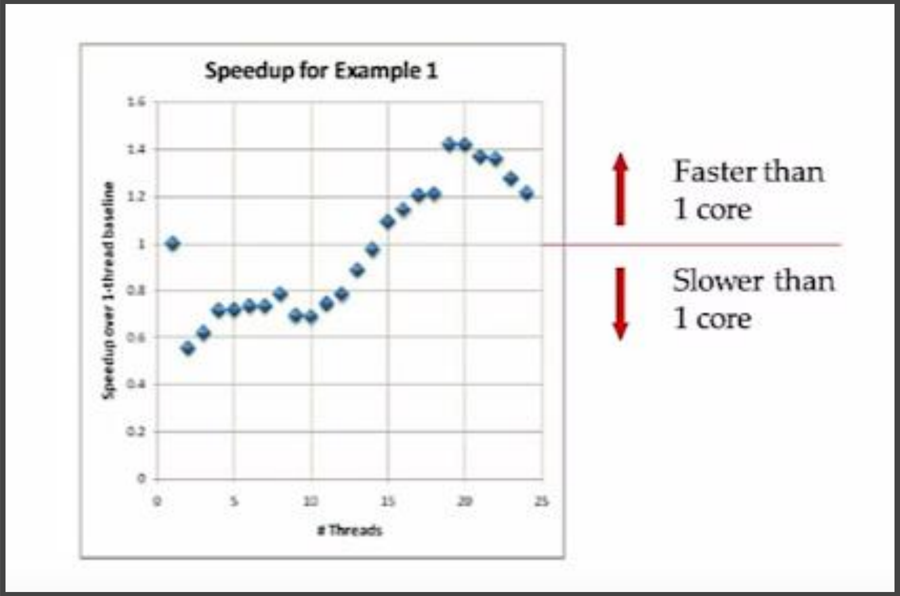
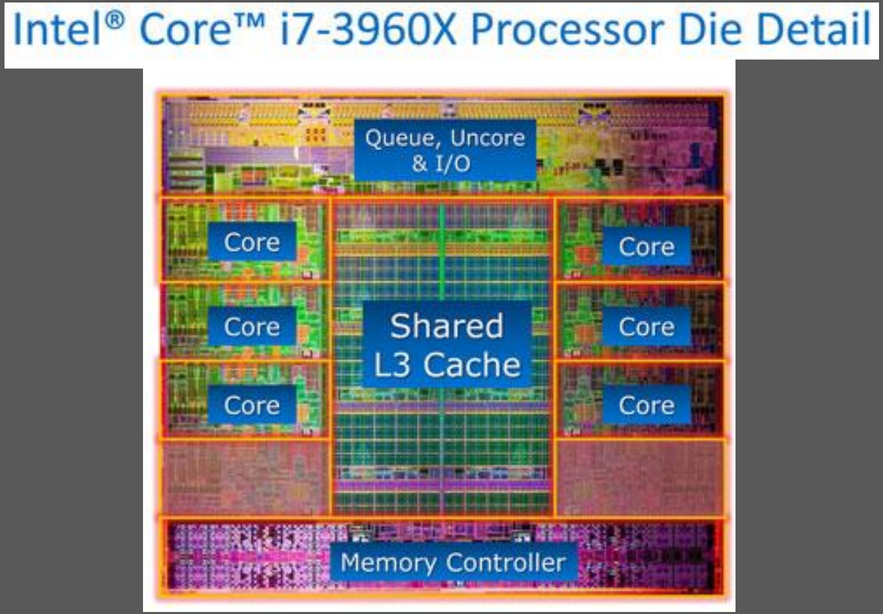


Image taken from a talk of Herb Sutter and Scott Meiers on concurrency and C++11.

1) Know your hardware: CPU

CPUs are fast:

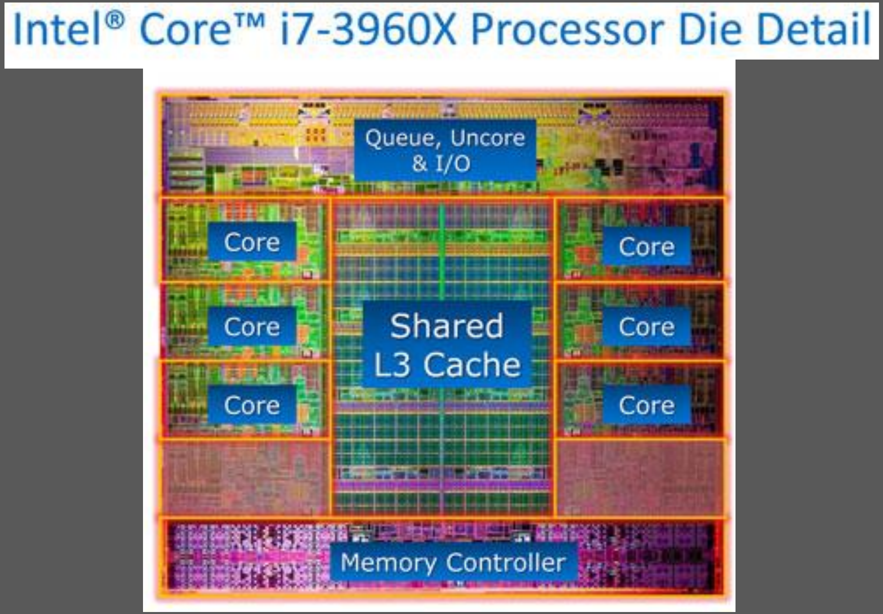
- 1.000.000.000 cycles/second
- 12+ cores per socket
- 3+ execution ports per core
- 36.000.000.000 instru/second



1) Know your hardware: CPU

CPUs are **too** fast:

- 1.000.000.000 cycles/second
- 12+ cores per socket
- 3+ execution ports per core
- 36.000.000.000 instru/second
- Waiting for data!



1) Know your hardware: CPU

CPUs are **too** fast:

- 1.000.000.000 cycles/second
- 12+ cores per socket
- 3+ execution ports per core
- 36.000.000.000 instru/second
- Waiting for data!

‘Pipeline’ of instructions.

BAD, aliasing is bad – prefetch

int v0 = 4;   auto& v1 = v0;   int* v2 = &v0;

BAD, concionals are bad - taking decisions beforehand

size_of(bool) == 8 bits   true or false

bool c = a && b   vs   char c = a & b;

good if it barely changes the prediction

BAD, virtualization is bad - vtable=alias

similar rule as conditionals

Multi-threading, not necessarily better!

Threads may share the cache. Caution!

1) Know your hardware: CPU

CPUs are **too** fast. Use them in advance!

a) Templates (compile time calculations)
   Well known factorial example.
   type traits can help you!

```
template<int n>
struct fibonacci
{
  static constexpr int value =
            fibonacci<n-1>::value +
            fibonacci<n-2>::value;
};

template<>
struct fibonacci<0>
{
  static constexpr value = 0;
};

template<>
struct fibonacci<1>
{
  static constexpr value = 1;
};
```

1) Know your hardware: CPU

CPUs are **too** fast. Use them in advance!

a) Templates (compile time calculations)
   Well known factorial example.
   type traits can help you!

b) `const` and `constexpr` (let the compiler know)
   A `constexpr` is known in <u>compile</u> time.
   A `const` is a value that it is not <u>expected</u> to change.

```
constexpr int pow (int base, int exp) noexcept
{
  auto result = 1;
  for(int i=0; i<exp; ++i) result *= base;

  return result;
}

constexpr auto numConds = 5;

std::array<int, pow(3, numConds)> results;
```

1) Know your hardware: CPU

CPUs are **too** fast. Use them in advance!

a) Templates (compile time calculations)
   Well known factorial example.
   type traits can help you!

b) `const` and `constexpr` (let the compiler know)
   A `constexpr` is known in compile time.
   A `const` is a value that it is not expected to change.

c) In C++17 we have `if-constrexpr`.

```
constexpr unsigned fibonacci(const unsigned x)
{
  return x <= 1 ?
    1 :
    fibonacci(x - 1) + fibonacci(x - 2);
}
           // or without the (A?B:C) operator…
constexpr unsigned fibonacci(const unsigned x)
{
  if constexpr(x <= 1)
    return 1;
  else
    return fibonacci(x - 1) + fibonacci(x - 2);
}

int main()
{
  return fibonacci(6);
}
```

1) Know your hardware: CPU

CPUs are **too** fast. Use them in advance!

a) Templates (compile time calculations)
  Well known factorial example.
  type traits can help you!

b) `const` and `constexpr` (let the compiler know)
  A `constexpr` is known in <u>compile</u> time.
  A `const` is a value that it is not <u>expected</u> to change.

c) In C++17 we have `if-constrexpr`.

d) `inline` functions (reduce misdirection)
  Avoid going to the pointer of the function.
  You will let the compiler to optimize more.

```
template <typename T>
inline const T& std::max(const T& a, const T& b)
{
  return a < b ? b : a;
}

// together with constant folding and propagation

inline float square(float x) { return x*x; }
inline float parabola(float x)
{
  return square(x) + 1.0f;
}

float a = parabola(2.0f);
float b = a + 1.0f;
…
float a = 5.0f;
float b = 6.0f;
```

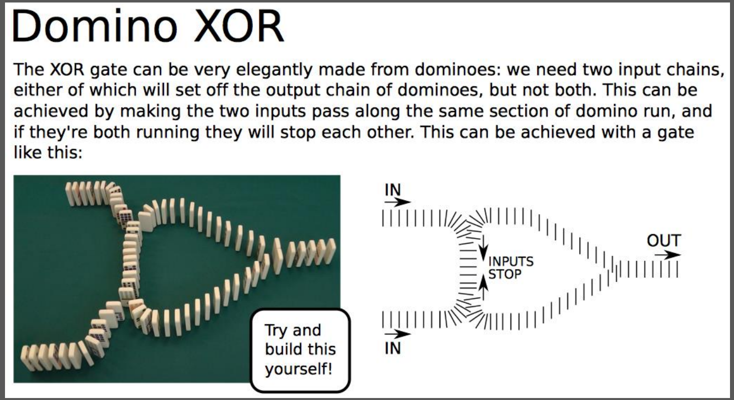
2) Know your hardware: Memory

CPUs are **too** fast:

Waiting for data!

“The only bad thing about theoretical computing is that there are no theoretical computers.”
Andy Thomason.

- Jeff Dean numbers.
- Cache speed & size comparison.



2) Know your hardware: Memory

Jeff Dean numbers!

```
Latency Comparison Numbers
--------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache

Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache

Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us

Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```


Note: These numbers are not perfectly accurate, and they don’t intent to be. The order of magnitude is accurate, though.

2) Know your hardware: Memory

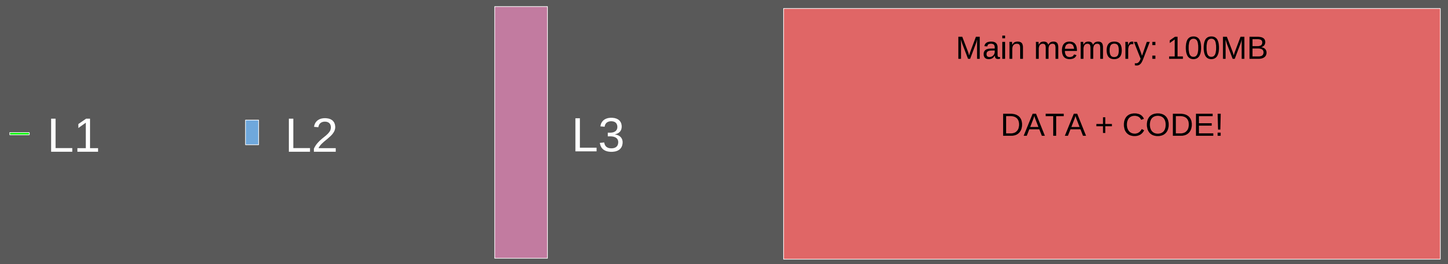
Speed comparison
(to scale, duh!)

Inspired in ‘Data-oriented desing and C++’. Mike Acton. Cppcon’14.

2) Know your hardware: Memory

CPUs are **too** fast. Waiting for data!

The level 1 **cache** is **too** small. The **main memory** is **BORINGLY** slow.



Data that fits the cache, compact data, and data contiguous in memory.

Code that fits the cache, compact code, and code contiguous in memory.

2) Know your hardware: Memory

Data that fits the cache, compact data, and data contiguous in memory.

Data alignment is **very** important in C++.
Store **together** variables that are used **together**.
Access data **sequentially**.

<u>Code</u> that fits the cache, compact <u>code</u>, and <u>code</u> contiguous in memory.

Store **together** functions that are used **together**.
Again: `inline` functions.

3) Know your numbers

Back-of-the-envelope calculations:

- Powers of 2.
- `size_of`: int, float, char...
- Size of cache (Jeff Dean numbers).
- Combinatory.
- Geometrical mathematics.
- <Name your own maths>.

2^0	**=**	**1**
2^1	=	2
2^2	=	4
2^3	=	8
2^4	**=**	**16**
2^5	=	32
2^6	=	64
2^7	=	128
2^8	**=**	**256**
2^9	=	512
2^{10}	=	1,024

Example: $2^{24} = ?$

$2^{24} = 2^{10} * 2^{10} * 2^4 =$

$\backsimeq 1000 * 1000 * 2^4 =$

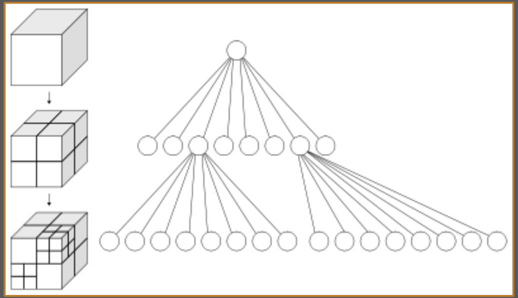
$= 1000 * 1000 * 16$

$2^{24} \backsimeq 16.000.000$

3) Know your numbers

Let's play.

- How to know if two spheres collide with each other?
  - Implement a method to detect collisions between two spheres.



Good data structures?

```
float radius;
float two_radius_square;
bool doCollide(point3d s1, point3d s2){
      float distance_square = (s1.x-s2.x)^2 + (s1.y-s2.y)^2 + (s1.z-s2.z)^2;
      return distance_square < two_radius_square;
}
```

Things to know, so far…

1) Know your hardware: **CPU**
   CPUs are TOO fast.
2) Know your hardware: **memory**
   L1 cache is fast, main memory is very slow.
3) Know your **numbers**
   Mathematics, powers of two…
4)
5)
6)
7)

4) Know your tools

- Language, C, C++, Java, Python, C#, openGLSL...

C++… What version are you using? Please, do modern C++

Learn! Cppcon, JavaOne, PyCon...

- Compiler, virtual machine? Just in time compilation?

C++… gcc? Mvisual? Clang? CLANG!

Do not help the compiler!

- You code for: hardware, peers, future you.

4) Know your tools

But… what would be Modern C++? (Snapshot from 2011 …. 6 years ago)

Modern C++ can be better… by default*.

AAA? (Almost Always Auto): the right type for every element.

Lambdas.

Smart Pointers: Helps a lot with RAII.

`nullptr`: Helps reducing bugs.

Range-based loops: Lets the compiler do its work.

move / &&: Helps handling the memory more efficient

But… there is more! C++14 and C++17

`constexpr`

`optional`

`variant`

…

* Taken from Herb Sutter´s blog.

4) Know your tools

Lot’s of the issues with CPU and Performance and Efficiency are due to bad coding.

a) Prefer `unique_ptr` to `shared_ptr`.
If you are only going to have ONE element… have one.

b) Always `const`. Try not to recalculate values.
As explained before, const can enable lots of optimizations!

c) Prefer initialization over assignment.

d) Use `explicit` keyword for constructors.
These undesired conversions waste time.

5) Know your algorithms

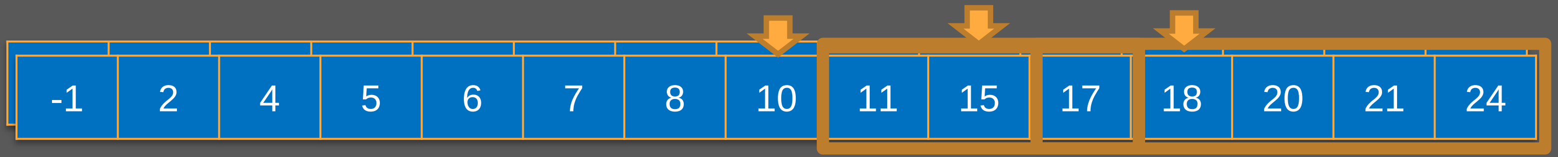
Computer science?

5) Know your algorithms

Efficiency through algorithms

“How much work does it take to do a task”

Improve efficiency by doing <u>less</u> work.




```
int find(vector<int> vec, int value);  //17?
```


Inspired in ‘Efficiency through algorithms and performance through data structures’. Chandler Carruth. Cppcon’14.

5) Know your algorithms

Example: Array sorting (based in a true story)

Do you know any good sorting algorithm?

Insertion-sort, Bubble-sort, Merge-sort, Radix-sort, Bucket-sort, Quick-sort

5) Know your algorithms

Example: Array sorting (based in a true story)

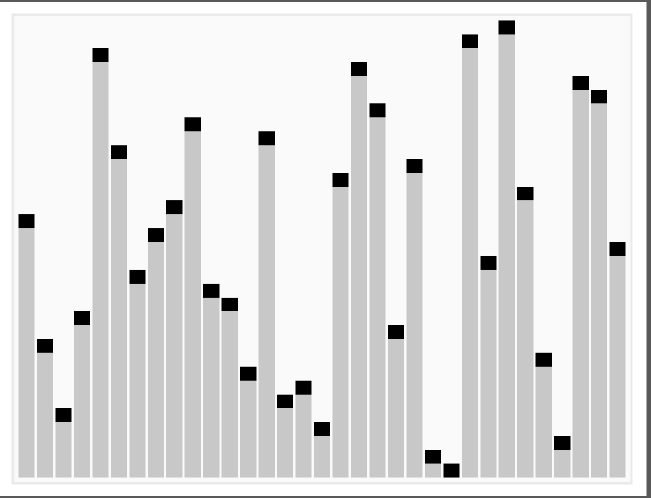
Do you know any good sorting algorithm?

3	1	2	5	10	6

n < p | p | ! (n < p)

1	2	**3**	5	10	6

Quick-sort! O(nlogn)



5) Know your algorithms

Example: Array sorting (based in a true story)

Do you know **a better** sorting algorithm? Quick-sort! O(nlogn)

Question: Did you think about corner cases? (repeated values)

5) Know your algorithms

Example: Array sorting (based in a true story)

Do you know **an <u>even</u> better** sorting algorithm? **3-way-quick-sort**! O(nlogn)

Question: Did you think about corner cases? (repeated values)

4	4	6	6	8	6	6	4	8	8	8	4	8	4	6

4	4	4	4	4

6

6	**6**	**6**	**6**

8	8	8	8	8

4

4	**4**	**4**	**4**

8

8	**8**	**8**	**8**

4	**4**	**4**	**4**	**4**	**6**	**6**	**6**	**6**	**6**	**8**	**8**	**8**	**8**	**8**

5) Know your algorithms

“Do less work by avoiding doing unnecessary work” C. Caruth

```
vector<X> f(int n){
   vector<X> result;
   for(int i=0; i<n; ++i)
      result.push_back(X(...));
   return result;
}
```

```
vector<X> f(int n){
   vector<X> result;
   result.reserve(n);
   for(int i=0; i<n; ++i)
      result.push_back(X(...));
   return result;
}
```


~~Inspired in~~ Copied from ‘Efficiency through algorithms and performance through data structures’. Chandler Carruth. Cppcon’14.

5) Know your algorithms

"Do less work by avoiding doing unnecessary work" C. Caruth

```
X *getX(string key,
        unordered_map<string,
                      unique_ptr<X>> &cache){

   if(cache[key])
       return cache[key].get();

   cache[key] = make_unique<X>(...);
   return cache[key].get;
}
```

```
X *getX(string key,
        unordered_map<string,
                      unique_ptr<X>> &cache){

   unique_ptr<X> &entry = cache[key];
   if(entry)
       return entry.get();

   entry = make_unique<X>(...);
   return entry.get;
}
```


~~Inspired in~~ Copied from 'Efficiency through algorithms and performance through data structures'. Chandler Carruth. Cppcon'14.

5) Know your algorithms

Efficiency through algorithms.

“How much work does it take to do a task”

Improve efficiency by doing less work.

Performance through data structures.

“How long does it take to your program to do an ammount of work”

Improve performance by faster doing your work.

Inspired in ‘Efficiency through algorithms and performance through data structures’. Chandler Carruth. Cppcon’14.

6) Know your data structures

Theory vs reality = performance!

"The goal of every program,
and of every component of those programs,
is to convert data from one form to another"
Mike Acton

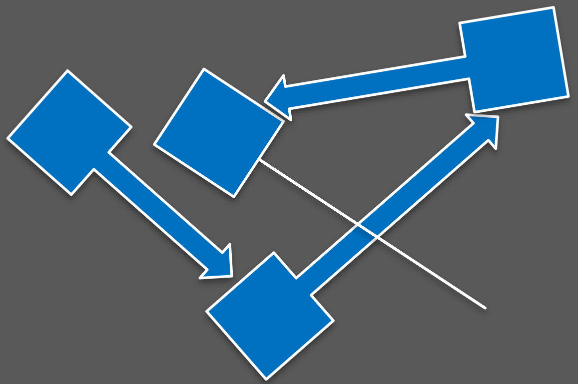
'Jack-of-all-trades' data structures?

Hybrid data structures?

Specialiced data structures?

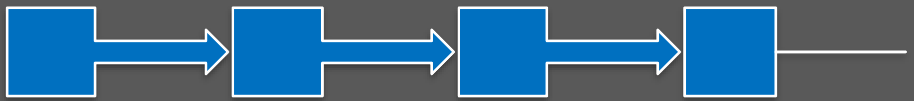
6) Know your data structures

Why you should **hate*** linked lists.



- Pointers, data aliasing.
- Every next element is a “cache miss”.
- Every element is allocated on his own.

- It *may* be good if you only traverse your list once.



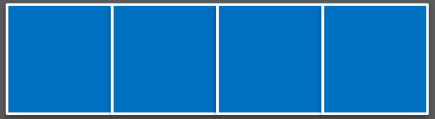
* You can quote me on this. And please, do challenge on this afterwards if you want.

6) Know your data structures

Why you should **love*** vectors and hash tables**.

- ‘Cache friendly’, compact, easy to handle, allocation.
- Stack, Queue, Linked list... Everything built upon an array.

- Good hash table:
  - Key-value pairs.
  - Contiguous in memory.
  - Good if both key and values are small.



* You can quote me on this.

** They are nothing more than a glorified array.

6) Know your data structures

AoS vs SoA (Array of Structs vs Struct of Arrays) (AKA: cache friendly)

```
struct cell_type{
   checking_box block;
   sample_type sample;
   other_stuff other;
};

vector<cell_type> cells;
//------------------------------------

point3d point;
for(int i=0; i<size; ++i)
{
   if(cells[i].block.check(point))
   {
      // do important things with cell
   }
}
```

[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]

6) Know your data structures

AoS vs SoA (Array of Structs vs Struct of Arrays) (AKA: cache friendly)

```
struct cell_type{
   checking_box block;
   sample_type sample;
   other_stuff other;
};

vector<cell_type> cells;
//--------------------------------------

point3d point;
for(int i=0; i<size; ++i)
{
   if(cells[i].block.check(point))
   {
      // do important things with cell
   }
}
```

[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]
[b	b	s	s	s	o	o]	[b	b	s	s	s	o	o]

6) Know your data structures

AoS vs SoA (Array of Structs vs Struct of Arrays) (AKA: cache friendly)

```
struct cell_type{
   checking_box block;
   sample_type sample;
   other_stuff other;
};
vector<cell_type> cells;
//------------------------------------
point3d point;
for(int i=0; i<size; ++i)
{
   if(cells[i].block.check(point))
   {
      // do important things with cells
   }
}
```

```
struct cell_type{
   sample_type sample;
   other_stuff other;
};
vector<checking_box> blocks;
vector<cell_type> cells;
//------------------------------------
point3d point;
for(int i=0; i<size; ++i)
{
   if(blocks[i].check(point))
   {
      auto& cell = cells[i];
      // do important things with cell
   }
}
```

6) Know your data structures

AoS vs SoA (Array of Structs vs Struct of Arrays) (AKA: cache friendly)

```
struct cell_type{
   sample_type sample;
   other_stuff other;
};
vector<checking_box> blocks;
vector<cell_type> cells;
//------------------------------------
point3d point;
for(int i=0; i<size; ++i)
{
   if(blocks[i].check(point))
   {
      auto& cell = cells[i];
      // do important things with cell
   }
}
```

[b	b]	[b	b]	[b	b]	[b	b]	[b	b]	[b	b]	[b	b]
[b	b]	[b	b]	[b	b]	[b	b]	[b	b]	[b	b]	[b	b]
[b	b]	[b	b]	[b	b]	[b	b]	[s	s	s	o	o]	[s
s	s	o	o]	[s	s	s	o	o]	[s	s	s	o	o]
[s	s	s	o	o]	[s	s	s	o	o]	[s	s	s	o
o]	[s	s	s	o	o]	[s	s	s	o	o]	[s	s	s
o	o]	[s	s	s	o	o]	[s	s	s	o	o]	[s	s
s	o	o]	[s	s	s	o	o]	[s	s	s	o	o]	[s
s	s	o	o]	[s	s	s	o	o]	[s	s	s	o	o]

6) Know your data structures

Efficiency through algorithms.

“How much work does it take to do a task”

Improve efficiency by doing less work.

Performance through data structures.

“How long does it take to your program to do an ammount of work”

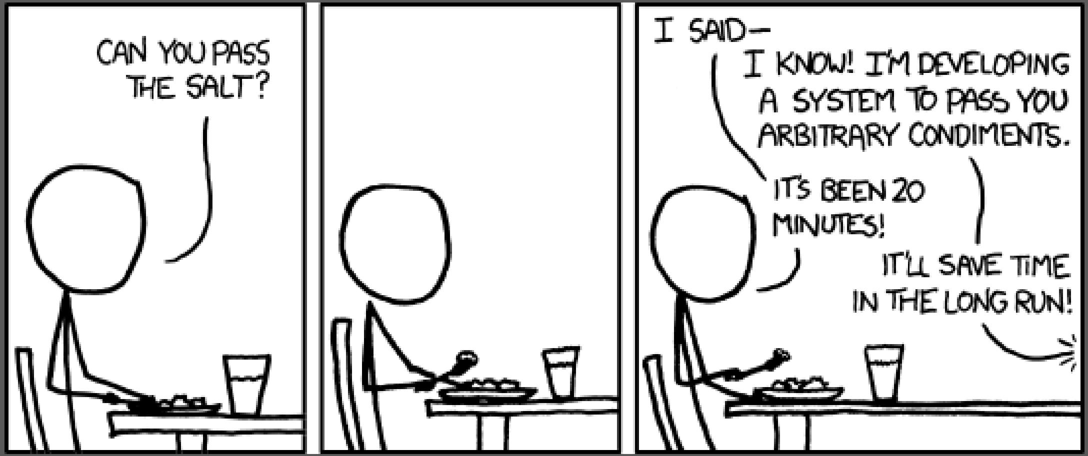
Improve performance by faster doing your work.

To sum up:

- Solve only ONE problem.
    - Check the data entropy.
- Data locality (cache!), *memory is slow!*
    - Array of structs vs struct of arrays (AoS vs SoA).

7) Know your problem

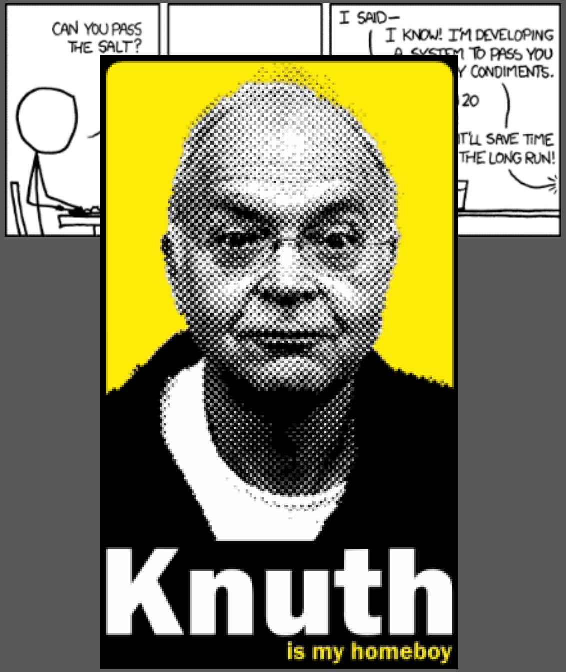
- Don’t solve a problem that you don’t have to solve.
- Use the common sense! Knowledge vs wisdom.
- Be careful regarding creating new problems.



7) Know your problem

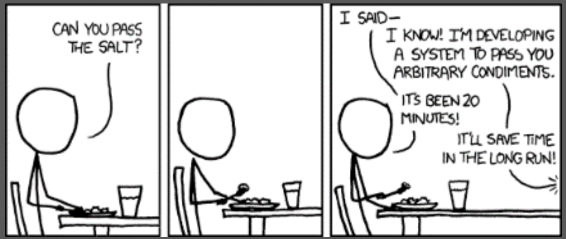
- Don’t solve a problem that you don’t have to solve.
- Use the common sense! Knowledge vs wisdom.
- Be careful regarding creating new problems.

- Don’t pre-optimize.
- Take measurements. Use ‘performance analysis’ tools.
- Look at the data. Size? Cache? Entropy?
- 20/80 rule.



7) Know your problem

- Don’t solve a problem that you don’t have to solve.
- Use the common sense! Knowledge vs wisdom.
- Be careful regarding creating new problems.



- Don’t pre-optimize.
- Take measurements. Use ‘performance analysis’ tools.
- Look at the data. Size? Cache? Entropy?
- 20/80 rule.

- Don’t get cozy. AKA Recognize your ignorance.
- ~~Go to~~ Watch talks of conferences.
- Ask. Try. Learn.

cppcon
the c++ conference

Seven topics to **know** to better use the CPU in your apps

1) Know your hardware: **CPU**
   CPUs are TOO fast.
2) Know your hardware: **memory**
   L1 cache is fast, main memory is very slow.
3) Know your **numbers**
   Mathematics, powers of two…
4) Know your **tools**
   Knowledge on language, compiler…
5) Know your **algorithms**
   Efficiency through algorithms, ‘do only the work you need’.
6) Know your **data structures**
   Performance through data structures, ‘make your work faster’.
7) Know your **problem**
   Use your common sense!

Thanks!

Any feedback?

Any questions?