¿Como salvar al mundo programando? ´
7 cosas que saber para ahorrar uso de CPU

Juanmi Huertas
R&D Software Engineer
Color and Imaging team
HP
juanmi.huertas@hp.com

30 de Noviembre
using std::cpp 2017, UC3M

Introducción

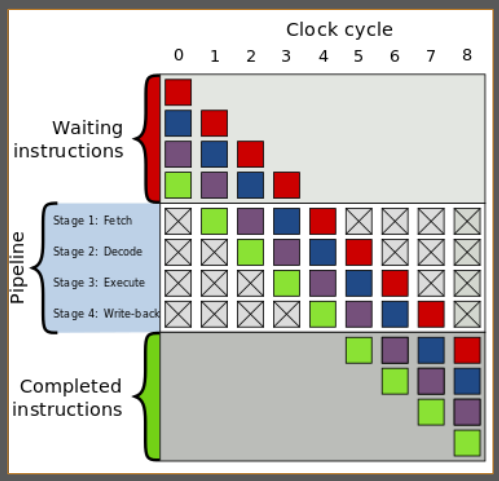
Siete cosas que **saber** para usar major la CPU en tus programas

Pero… ¿por qué nos preocupamos de la CPU?

1) Conoce tu hardware: CPU

"La única cosa mala sobre la computación teórica es que no hay ordenadores teóricos."
Andy Thomason

Pipeline de la CPU

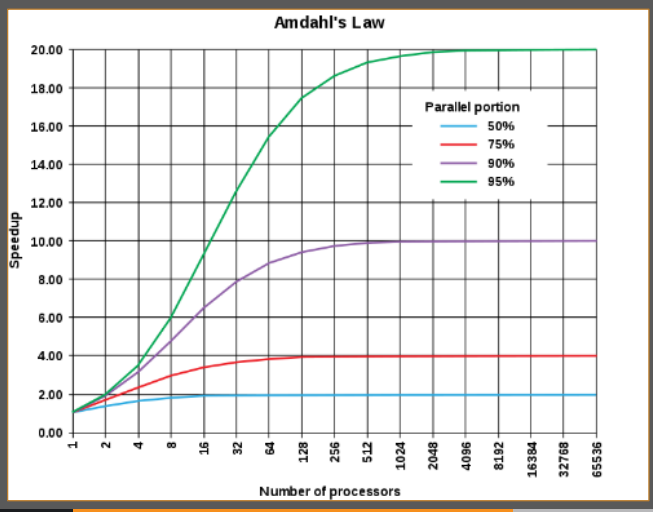


1) Conoce tu hardware: CPU

"La única cosa mala sobre la computación teórica es que no hay ordenadores teóricos."
Andy Thomason

Pipeline de la CPU

Concurrencia.



1) Conoce tu hardware: CPU

"La única cosa mala sobre la computación teórica es que no hay ordenadores teóricos."
Andy Thomason

Pipeline de la CPU

¿Concurrencia?

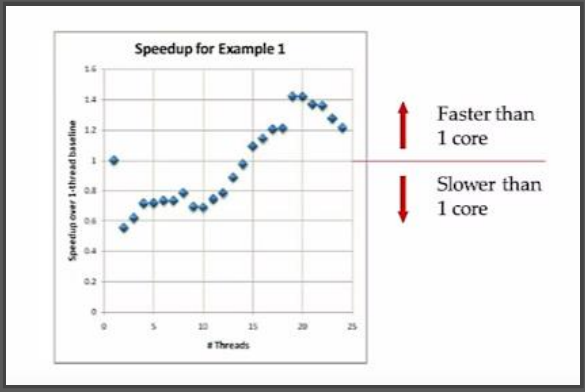
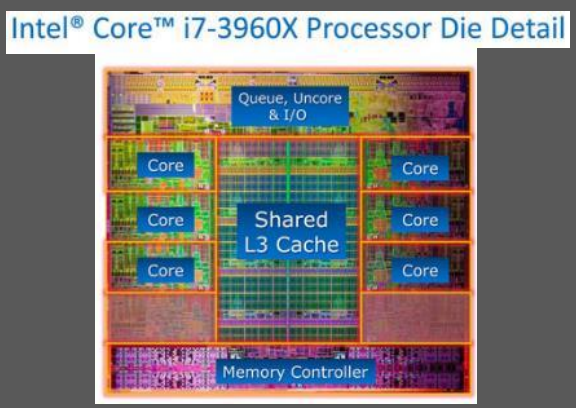


Imagen tomada en charlas de Herb Sutter y Scott Meiers sobre concurrencia y C++11.

1) Conoce tu hardware: CPU

Los CPUs son rápidos:

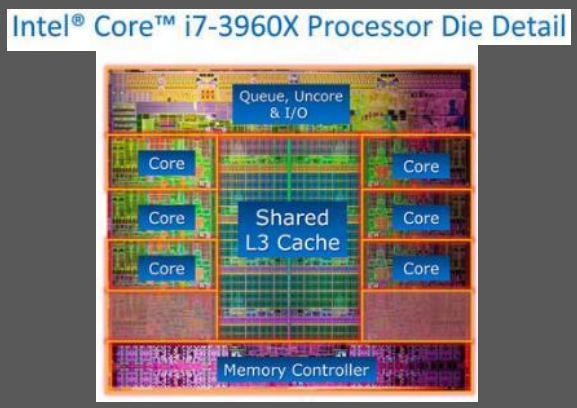
- 1.000.000.000 ciclos/segundo
- 12+ cores por socket
- 3+ puertos de ejecución por core
- 36.000.000.000 instru/segundo



1) Conoce tu hardware: CPU

Los CPUs son **demasiados** rápidos:

- 1.000.000.000 ciclos/segundo
- 12+ cores por socket
- 3+ puertos de ejecución por core
- 36.000.000.000 instru/Segundo
- ¡Esperando los datos!



1) Conoce tu hardware: CPU

Los CPUs son **demasiado** rápidos:

- 1.000.000.000 ciclos/segundo
- 12+ cores por socket
- 3+ puertos de ejecución por core
- 36.000.000.000 instru/segundo
- ¡Esperando los datos!

'Pipeline' de instrucciones.

Pipelíne de instrucciones.

MALO, aliasing – prefetch

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

MALO, condicionales – taking decisions beforehand

```
size_of(bool) == 8 bits        true or false
bool c = a && b       vs       char c = a & b;
```

Bueno si no cambia la predicción

MALO, virtualization - vtable=alias

Mismas reglas que los condicionales

Multi-threading, ¡no es siempre mejor!

Multi-threading, ¡no es siempre mejor!

1) Conoce tu hardware: CPU

Los CPUs son rápidos. ¡Usalos!

a) Templates (tiempo de compilación).
Los type traits pueden ayudarte!

```
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) Conoce tu hardware: CPU

Los CPUs son rápidos. ¡Usalos!

a) Templates (tiempo de compilación).
Los type traits pueden ayudarte!

b) `const` y `constexpr` (avisa al compilador)
Un `constexpr` es conocida en tiempo de compilación.
Un `const` es un valor que "cambiará".

```
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) Conoce tu hardware: CPU

Los CPUs son rápidos. ¡Usalos!

a) Templates (tiempo de compilación).
Los type traits pueden ayudarte!

b) `const` y `constexpr` (avisa al compilador)
Un `constexpr` es conocida en tiempo de compilación.
Un `const` es un valor que "cambiará".

c) En C++17 temenos `if-constrexpr`.

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

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

11/44

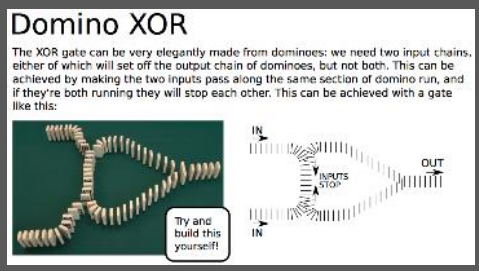
2) Conoce tu hardware: Memoria

Los CPUs son **muy rápidos**:

¡Esperando los datos!

"La única cosa mala de la computación teórica es que no hay ordenadores teóricos."
Andy Thomason.

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



12/44

2) Conoce tu hardware: Memoria

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
```


Nota: Estos números no son perfectamente precisos, y no lo pretenden. Los órdenes de magnitude son fiables.

2) Conoce tu hardware: Memoria

Comparación de velocidad
(A escala…)

Inspirado en ‘Data-oriented desing and C++’. Mike Acton. Cppcon’14.

2) Conoce tu hardware: Memoria

La CPUs es **muy** rápida. ¡Esperando los datos!

La cache de nivel 1 es **muy** pequeña. La **memoria principal** es **muy** lenta.

Datos que entren en cache, datos compactos, y datos contiguos en memoria.
Código que entre en cache, código compacto, y código contiguo en memoria.

15/44

2) Conoce tu hardware: Memoria

Datos que entren en cache, datos compactos, y datos contiguos en memoria.

El alineamiento de datos es **muy** importante en C++.
Almacena **juntas** variables que sean usadas **juntas**.
Accede tus datos **secuencialmente**.

Código que entre en cache, código compacto, y código contiguo en memoria.

Almacena **juntas** funciones que sean usadas **juntas**.

16/44

3) Conoce tus números

Calculos automáticos:

- Potencias de dos.
- `size_of`: int, float, char...
- Tamaño de la cache (Jeff Dean numbers).
- Combinatoria.
- Geometría.
- <Inserta aquí más matemáticas>.

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

Ejemplo: 2^{24} = ?

$2^{24} = 2^{10} * 2^{10} * 2^4 =$
$\backsimeq 1000 * 1000 * 2^4 =$
$= 1000 * 1000 * 16$

$2^{24} \backsimeq 16.000.000$

Ejemplo: 131.072 = ?

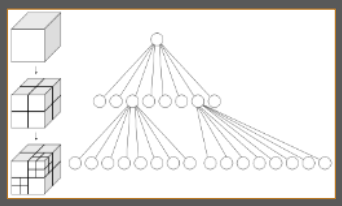
$131.072 \backsimeq 131 * 1000 =$
$= 131 * 2^{10} \backsimeq 128 * 2^{10} =$
$= 2^7 * 2^{10} = 2^{17}$

$131.072 \backsimeq 2^{17}$

3) Conoce tus números

Juguemos.

- ¿Cómo saber si dos esferas colisionan entre ellas?



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;
}
```

Cosas que conocer, por ahora…

1) Conoce tu hardware: **CPU**
   Las CPUs son MUY rápidas.
2) Conoce tu hardware: **memoria**
   L1 cache es rápida, memoria principal es muy lenta.
3) Conoce tus **números**
   Matemáticas, potencias de dos…
4)
5)
6)
7)

4) Conoce tus herramientas

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

C++… ¿Qué versión estás usando? Usa C++ moderno.

¡Aprende! Cppcon, JavaOne, PyCon...

- Compiler, virtual machine? Just in time compilation?

C++… gcc? Mvisual? Clang? CLANG!

- Programas para: hardware, colegas, tu futuro tú.

4) Conoce tus herramientas

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 + adds intent.
`nullptr`: Helps reducing bugs + adds intent.
Range-based loops: Lets the compiler do its work + adds intent.
move / &&: Helps handling the memory more efficiently.

But wait… there is more! C++14 and C++17

`constexpr`
`optional`
`variant`
…

* Tomado del blog de Herb Sutter.

4) Conoce tus herramientas

Muchos problemas de rendimiento y eficiencia se deben a mal código.

a) Prefiere `unique_ptr` a `shared_ptr`.
Si solo tienes UN elemento… ten únicamente UNO.

b) Siempre `const`. Intenta no recalcular valores.
`const` puede ayudar a reducir errores.

c) Prefiere inicialización sobre asignación.
No dejas valores sin inicializar y es más eficiente.

d) Usa `explicit` para constructores.
Las conversiones no deseadas gastan recursos.

5) Conoce tus algoritmos

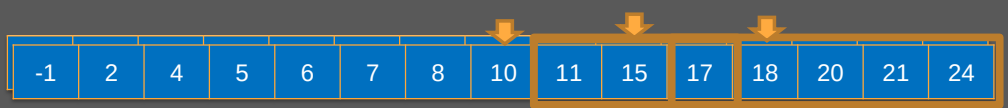
¿Computer science?

5) Conoce tus algoritmos

Eficiencia con algoritmos

“Cuánto trabajo hace falta para resolver una tarea”

Mejora la eficiencia hacienda menos trabajo.




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

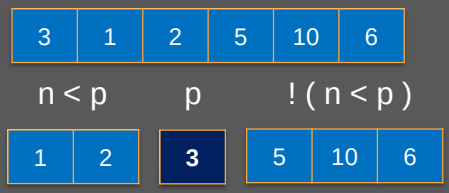

Inspired in ‘Efficiency through algorithms and performance through data structures’. Chandler Carruth. Cppcon’14.

5) Conoce tus algoritmos

Ejemplo: Ordenar un array (basado en una historia real)

¿Conocéis un algoritmo de ordenación bueno? Quick-sort!

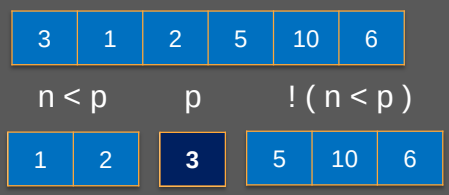
Bubble-sort, insertion-sort, radix-sort, heap-sort, merge-sort, quick-sort…

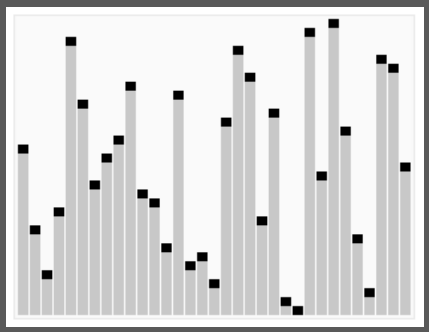


5) Conoce tus algoritmos

Ejemplo: Ordenar un array (basado en una historia real)

¿Conocéis un algoritmo de ordenación bueno? Quick-sort!





5) Conoce tus algoritmos

Ejemplo: Ordenar un array (basado en una historia real)

¿Conocéis un algoritmo de ordenación **mejor**? quick-sort! O(nlogn)

Pregunta: ¿Casos extremos? (valores repetidos)

5) Conoce tus algoritmos

Ejemplo: Ordenar un array (basado en una historia real)

¿Conocéis un algoritmo de ordenación **aun mejor**? **3-way-quick-sort**! O(nlogn)

Pregunta: ¿Casos extremos? (valores repetidos)

5) Conoce tus algoritmos

"Haz menos trabajo no hacienda el trabajo innecesario" 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;
}
```

29/44

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

5) Conoce tus algoritmos

"Haz menos trabajo no hacienda el trabajo innecesario" 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;
}
```

30/44

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

5) Conoce tus algoritmos

Eficiencia con algoritmos.

"Cuánto trabajo hace falta para resolver una tarea"

Mejora la eficiencia hacienda menos trabajo.

Rendimiento con estructuras de datos.

"Cuánto tiempo necesita tru programa para hacer un trabajo"

Mejora el rendimiento haciedo el trabajo más rápido.

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

6) Conoce tus estructuras de datos

Teoría vs realidad = ¡rendimiento!

"El objetivo de cada programa, y de cada componente en dicho programa, es convertir datos de una forma a otra"
Mike Acton

¿Estructura de datos comodín?

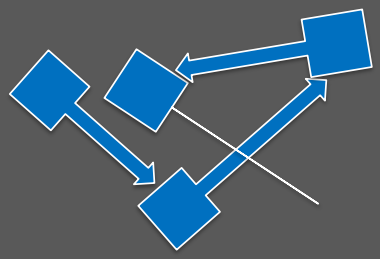
¿Estructura de datos híbrida?

¿Estructura de datos especializada?

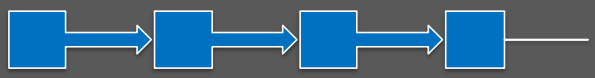
6) Conoce tus estructuras de datos

Por qué podrías **odiar** las listas enlazadas.

- Punteros, data aliasing.
- Cada siguiente elemento es un “cache miss”.
- Cada elemento se reserva individualmente.



- *Podría* ser bueno si solo se atraviesa la lista una vez.



6) Conoce tus estructuras de datos

Por qué podrías **amar** a los vectores y a las tablas hash*.

- ‘Cache friendly’, compactos, fácil de manejar, reserva de memoria.
- Stack, Queue, Linked list... Todo se puede hacer sobre un array.

- Buena tabla hash:
  - Par llave-valor.
  - Contíguo en memoria.
  - Llaves y valores pequeños.

* Podrían ser entendidos como un “array” glorificado..

6) Conoce tus estructuras de datos

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]

35/44

6) Conoce tus estructuras de datos

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]

36/44

6) Conoce tus estructuras de datos

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
   }
```

37/44

6) Conoce tus estructuras de datos

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]

38/44

6) Conoce tus estructuras de datos

Eficiencia con algoritmos.

"Cuánto trabajo hace falta para resolver una tarea"

Mejora la eficiencia hacienda menos trabajo.

Rendimiento con estructuras de datos.

"Cuánto tiempo necesita tu programa para hacer un trabajo"

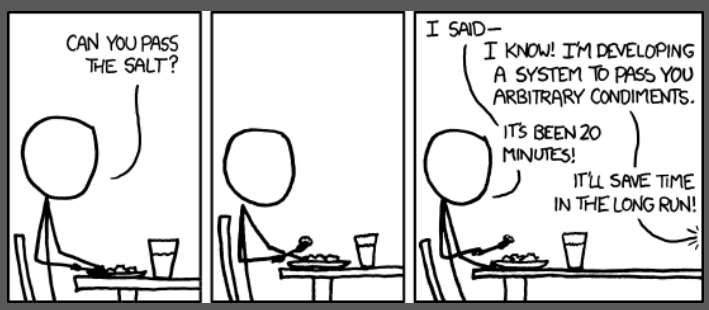
Mejora el rendimiento haciedo el trabajo más rápido.

Resumiendo:

- Resuelve UN problema.
  - Comprueba la entropía de tus datos.
- Localidad de los datos (¡cache!), ¡la*memoria es lenta!*
  - Array of structs vs struct of arrays (AoS vs SoA).

7) Conoce tu problema

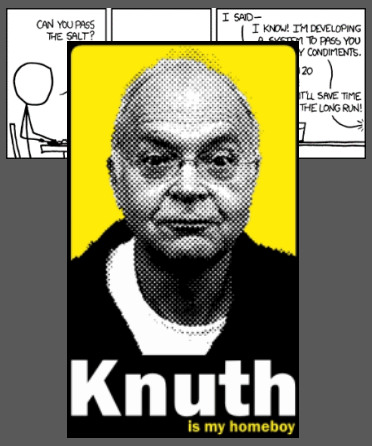
- No resuelvas lo que no tienes que resolver.
- ¡Usa el sentido común! Conocimiento vs sabiduria.
- Ten cuidado al crear nuevos problemas.



7) Conoce tu problema

- No resuelvas lo que no tienes que resolver.
- ¡Usa el sentido común! Conocimiento vs sabiduria.
- Ten cuidado al crear nuevos problemas.

- No pre-optimices.
- Toma medidas. Usa herramientas de análisis de rendimiento.
- Mira los datos. ¿Tamaño? ¿Cache? ¿Enrtropía?
- Regla del 20/80.



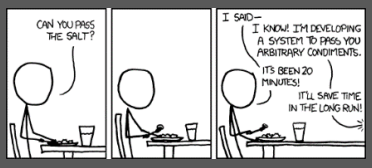
41/44

7) Conoce tu problema

- No resuelvas lo que no tienes que resolver.
- ¡Usa el sentido común! Conocimiento vs sabiduria.
- Ten cuidado al crear nuevos problemas.

- No pre-optimices.
- Toma medidas. Usa herramientas de análisis de rendimiento.
- Mira los datos. ¿Tamaño? ¿Cache? ¿Enrtropía?
- Regla del 20/80.

- No te acomodes. AKA Reconoce tu ignorancia.
- ~~Asiste a~~ Ve charlas de conferencias.
- Pregunta. Prueba. Aprende.



cppcon
the c++ conference

42/44

Sietes cosas que **conocer** para usar major la CPU en tus apps

1) Conoce tu hardware: **CPU**
   Los CPUs son MUY rápidos.
2) Conoce tu hardware: **memoria**
   L1 cache es rápida, memoria principal es muy lenta.
3) Conoce tus **números**
   Matemáticas, potencias de dos…
4) Conoce tus **herramientas**
   Conoce tu/s lenguaje/s y tu/s herramienta/s…
5) Conoce tus **algoritmos**
   La eficiencia viene de los algoritmos, 'solo haz el trabajo que necesites'.
6) Conoce tus **estructuras de datos**
   El rendimiento viene de las estructuras de datos, 'haz el trabajo más rápido'.
7) Conoce tu **problema**
   ¡Usa el sentido común!

43/44

¡Gracias!

¿Algún feedback?

¿Algunas preguntas?

44/44