“What makes a program fast” or “How CPUs are complex beasts”

Jordi Olivares

January 7, 2020

Section 1

What is this talk about

Recent articles

You might have seen a recent article/talk regarding CPUs and how they affect our code.

- C Is Not a Low-Level Language (2018) - Hacker News
- CPU design effects - Jakub Beránek - Meeting C++ 2019 - r/cpp

Recent articles

You might have seen a recent article/talk regarding CPUs and how they affect our code.

- C Is Not a Low-Level Language (2018) - Hacker News
- CPU design effects - Jakub Beránek - Meeting C++ 2019 - r/cpp

This talk is an extension of that

A basic concept of CPUs

In order to understand CPUs and how they affect us we need to understand a couple of simple concepts:

A basic concept of CPUs

In order to understand CPUs and how they affect us we need to understand a couple of simple concepts:

- A CPU is currently a Hardware VM

A basic concept of CPUs

In order to understand CPUs and how they affect us we need to understand a couple of simple concepts:

- A CPU is currently a Hardware VM
- Memory is really slow compared to cache

Latency scale

System Event	**Actual Latency**	**Scaled Latency**
CPU Cycle	0.4 ns	1 s
L1 Cache access	0.9 ns	2 s
L2 Cache access	2.8 ns	7 s
L3 Cache access	28 ns	1 min
Main Memory access	$\sim$100 ns	**4 min**

Section 2

Let's explore CPU optimizations

The code to optimize

```python
sum(
    object.key
    for object in list_of_objects
    if object.key < 0.5
)
```

A note about these benchmarks

In practice we will do all the benchmarks in C++ as it will allow us control over the code generated.

All of these methods are CPU dependent though and should be useful regardless of programming language used.

The object itself that we will measure

```
struct Object {
    float key;
    int filler1;
    int filler2;
    int filler3;
    int filler4;
    int filler5;
    int filler6;
    int filler7;
};
```

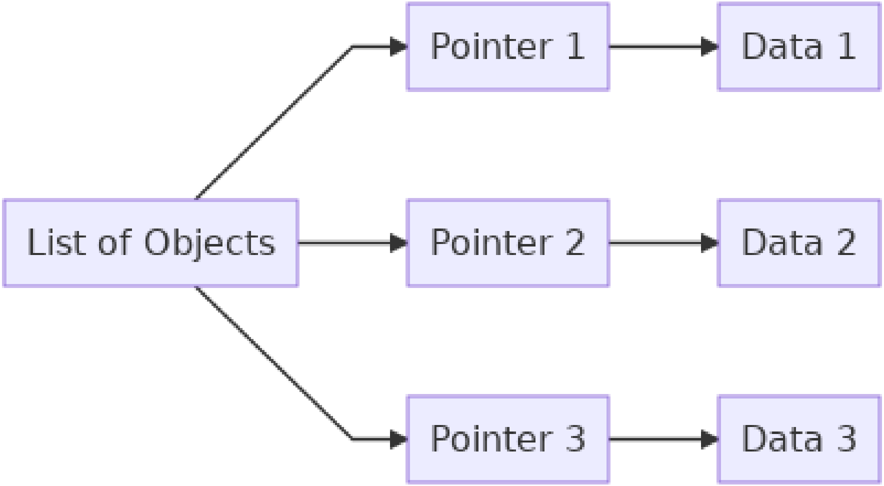
Section 3

The naive version

The code

```
static void naiveVersion(benchmark::State &state) {
    auto elements = naiveObjects(state.range(0));
    for (auto _ : state) {
        float total = 0;
        for (const auto &element : elements) {
            if (element->key < 0.5) {
                total += element->key;
            }
        }
        benchmark::DoNotOptimize(total);
    }
}
```

How is it structured in memory?



The measurements

Benchmark	Elements	Time	Iterations
Naive	8	7 ns	100597298
Naive	64	53 ns	12658713
Naive	512	419 ns	1702633
Naive	4096	15597 ns	44802
Naive	32768	167514 ns	4132
Naive	65536	342738 ns	2046

Section 4

Data Local version (Removing one indirection)

The code

```
static void localVersion(benchmark::State &state) {
    auto elements = localObjects(state.range(0));
    for (auto _ : state) {
        float total = 0;
        for (const auto &element : elements) {
            if (element.key < 0.5) {
                total += element.key;
            }
        }
        benchmark::DoNotOptimize(total);
    }
}
```

How is the memory laid out?

The results

Benchmark	Elements	Time	Iterations
Local	8	6.22 ns	98679509
Local	64	45.4 ns	12803676
Local	512	425 ns	1588957
Local	4096	14199 ns	49682
Local	32768	147016 ns	4733
Local	65536	295354 ns	2378

An ~**14%** decrease in time spent!

- In C/C++ it's a matter of just using `structs`

An ~**14%** decrease in time spent!

- In C/C++ it's a matter of just using `structs`
- In Java you can't do this, it's in progress with Project Valhalla :(

An ~**14%** decrease in time spent!

- In C/C++ it's a matter of just using `structs`
- In Java you can't do this, it's in progress with Project Valhalla :(
- In C# they are the so-called `structs` too

An ~**14%** decrease in time spent!

- In C/C++ it's a matter of just using `structs`
- In Java you can't do this, it's in progress with Project Valhalla :(
- In C# they are the so-called `structs` too
- Usually called Value Types, so search for them in your preferred language

Section 5

Columnar version

The idea

- We are storing “rows” of objects

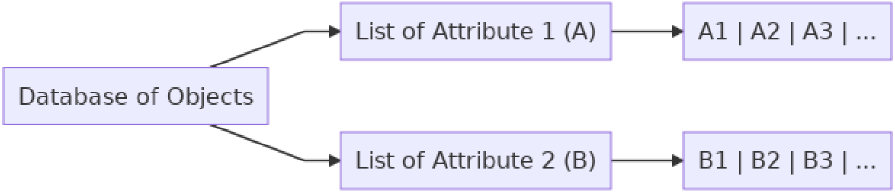
The idea

- We are storing “rows” of objects
- What if we store one column per attribute instead?

The idea

- We are storing “rows” of objects
- What if we store one column per attribute instead?
- We only have to iterate all the `key` attribute instances that are packed next to each other!

The memory layout



The code

```
static void columnarVersion(benchmark::State &state) {
    auto elements = columnarObjects(state.range(0));
    auto &column = elements.getColumn<0>();
    for (auto _ : state) {
        float total = 0;
        for (const auto element : column) {
            if (element < 0.5) {
                total += element;
            }
        }
        benchmark::DoNotOptimize(total);
    }
}
```

The results

Benchmark	Elements	Time	Iterations
Columnar	8	6.34 ns	111646812
Columnar	64	48.9 ns	13275731
Columnar	512	422 ns	1573965
Columnar	4096	12675 ns	55300
Columnar	32768	139632 ns	4997
Columnar	65536	287098 ns	2449

. . . just a **3%** decrease over the previous results

Section 6

Pros and cons

Pros:

- Maximizes data locality of an attribute

Pros:

- Maximizes data locality of an attribute
- Allows vectorization of code if needed (will be shown)

Cons:

- Requires a relatively large refactor of codebases as you have to use a "database" instead of a list of objects

Cons:

- Requires a relatively large refactor of codebases as you have to use a "database" instead of a list of objects
- Will couple database with object type

This is great!

This is great!

But can we make it *faster*?

Section 7

Let's talk about the CPU pipeline

What is it

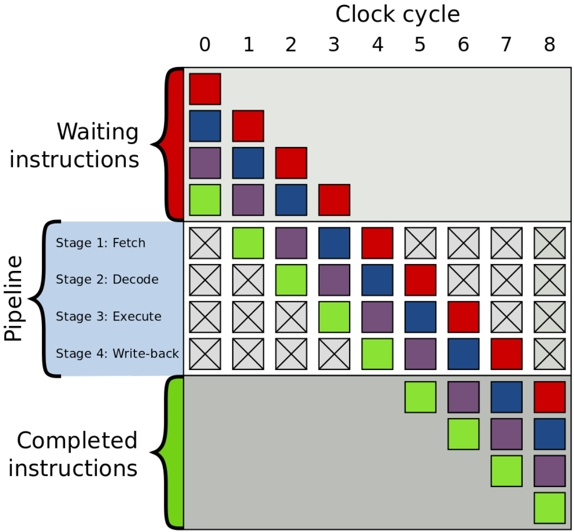


Figure 1: Pipeline

Branches and how we deal with them

Usually CPUs will do a combination of all of these:

- Stall the pipeline a bit by inserting a `NOP` in some steps and waiting until results can be used

Branches and how we deal with them

Usually CPUs will do a combination of all of these:

- Stall the pipeline a bit by inserting a `NOP` in some steps and waiting until results can be used
- Predict which branch we usually take and accordingly pipeline their instructions in order to avoid stalling

Branches and how we deal with them

Usually CPUs will do a combination of all of these:

- Stall the pipeline a bit by inserting a `NOP` in some steps and waiting until results can be used
- Predict which branch we usually take and accordingly pipeline their instructions in order to avoid stalling
  - If we fail to predict then we have to flush all the pipelined instructions and rewind back to where we should be.

Branches and how we deal with them

Usually CPUs will do a combination of all of these:

- Stall the pipeline a bit by inserting a `NOP` in some steps and waiting until results can be used
- Predict which branch we usually take and accordingly pipeline their instructions in order to avoid stalling
  - If we fail to predict then we have to flush all the pipelined instructions and rewind back to where we should be.
  - Current Intel CPUs have upwards of 14/16 stages so a flush can be *VERY* expensive.

Let's go back to the naive version

```
static void naiveVersion(benchmark::State &state) {
    auto elements = makeDataElements(state.range(0));
    for (auto _ : state) {
        float total = 0;
        for (const auto &element : elements) {
            if (element->key < 0.5) {
                total += element->key;
            }
        }
        benchmark::DoNotOptimize(total);
    }
}
```

The branch prediction will continuously fail during the `if (element->key < 0.5)` as the data is random causing continuous flushes and/or stalls!

This can be solved if we sort the data before going through the elements.

The results

Benchmark	Elements	Time	Iterations
Naive + Sorted	32768	49329 ns	14257
Naive + Sorted	65536	120139 ns	6129
Local + Sorted	32768	36984 ns	18995
Local + Sorted	65536	74605 ns	9569
Columnar + Sorted	32768	24906 ns	27385
Columnar + Sorted	65536	50529 ns	13968

The gains are pretty substantial!

We've decreased the time taken by

Type of data	Before	After	Reduction %
Naive version	342738 ns	120139 ns	~65%
Local version	295354 ns	74605 ns	~75%
Columnar version	287098 ns	50529 ns	~82%

A caveat for this particular case

Current CPUs have assembly instructions for doing conditional moves that effectively nullifies the difference between sorted and unsorted sets in our example.

These are called conditional moves and exist both for integers and floats. Oddly enough, the compilers are only using the integer ones and disregarding the assembly codes for floating point conditional moves.

This example wanted to showcase the branch prediction thus the reason for using floats and not ints.

Section 8

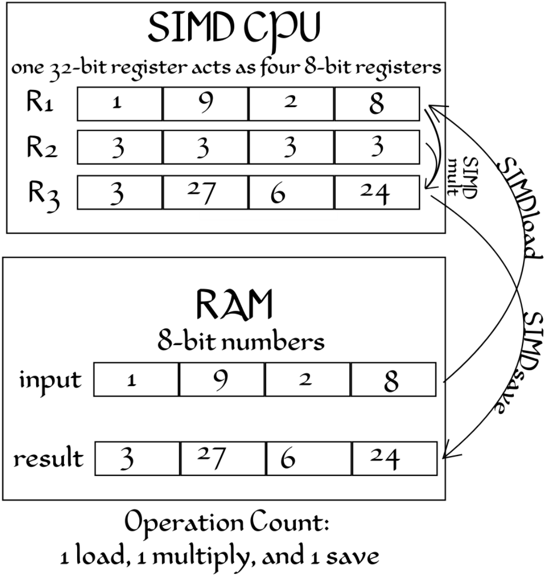
One final trick with SIMD

What is it

Single **I**nstruction **M**ultiple **D**ata

Also called *Vectorization*

How it works


SIMD CPU
one 32-bit register acts as four 8-bit registers
R1
1
9
2
8
R2
3
3
3
3
R3
3
27
6
24
SIMD mult
SIMDload
SIMDsave
RAM
8-bit numbers
input
1
9
2
8
result
3
27
6
24
Operation Count:
1 load, 1 multiply, and 1 save

How can I use it in my code?

- Can only be done with the sorted columnar data version

How can I use it in my code?

- Can only be done with the sorted columnar data version
- It's CPU dependent (ARM NEON/SVE, x86 MMX/SSE family/AVX family)

How can I use it in my code?

- Can only be done with the sorted columnar data version
- It's CPU dependent (ARM NEON/SVE, x86 MMX/SSE family/AVX family)
- Is out of reach for Java as you absolutely rely on the JVM JIT capabilities or creating a JNI wrapper

How can I use it in my code?

- Can only be done with the sorted columnar data version
- It's CPU dependent (ARM NEON/SVE, x86 MMX/SSE family/AVX family)
- Is out of reach for Java as you absolutely rely on the JVM JIT capabilities or creating a JNI wrapper
- C# has the capability by using the `System.Numerics` package

How can I use it in my code?

- Can only be done with the sorted columnar data version
- It's CPU dependent (ARM NEON/SVE, x86 MMX/SSE family/AVX family)
- Is out of reach for Java as you absolutely rely on the JVM JIT capabilities or creating a JNI wrapper
- C# has the capability by using the `System.Numerics` package
- C/C++ has CPU intrinsics available, but I personally use `xsimd` to be CPU independent

The final results

Benchmark	Elements	Time	Iterations
Naive	65536	342738 ns	2046
Local	65536	295354 ns	2378
Columnar	65536	287098 ns	2449
Naive + Sorted	65536	120139 ns	6129
Local + Sorted	65536	74605 ns	9569
Columnar + Sorted	65536	50529 ns	13968
Columnar + Sorted + SIMD	65536	16622 ns	41787

The final results

Benchmark	Elements	Time	Iterations
Naive	65536	342738 ns	2046
Local	65536	295354 ns	2378
Columnar	65536	287098 ns	2449
Naive + Sorted	65536	120139 ns	6129
Local + Sorted	65536	74605 ns	9569
Columnar + Sorted	65536	50529 ns	13968
Columnar + Sorted + SIMD	65536	16622 ns	41787

We've achieved a:

- ~**67%** reduction of time spent compared to the sorted columnar version, or a **3x** increase in throughput

The final results

Benchmark	Elements	Time	Iterations
Naive	65536	342738 ns	2046
Local	65536	295354 ns	2378
Columnar	65536	287098 ns	2449
Naive + Sorted	65536	120139 ns	6129
Local + Sorted	65536	74605 ns	9569
Columnar + Sorted	65536	50529 ns	13968
Columnar + Sorted + SIMD	65536	16622 ns	41787

We've achieved a:

- $\sim$**67%** reduction of time spent compared to the sorted columnar version, or a **3x** increase in throughput
- $\sim$**95%** reduction of time spent compared to the original naive version, or a **20x** increase in throughput

A final note about all of this

- The performance increase is noticeable in high number of elements

A final note about all of this

- The performance increase is noticeable in high number of elements
- In low numbers almost none of this matters

A final note about all of this

- The performance increase is noticeable in high number of elements
- In low numbers almost none of this matters
    - Except SIMD for obvious reasons

Benchmark	Elements	Time	Iterations
Naive	512	422 ns	1699728
Local	512	430 ns	1666707
Columnar	512	454 ns	1495578
Naive + Sorted	512	445 ns	1596756
Local + Sorted	512	554 ns	1265820
Columnar + Sorted	512	400 ns	1874843
Columnar + Sorted + SIMD	512	133 ns	5061195

Conclusion

- Only do this if you absolutely need the performance.

Conclusion

- Only do this if you absolutely need the performance.
- Most of the time don't even bother being smart about code, memory layout, or CPU specifics until you reach a certain size