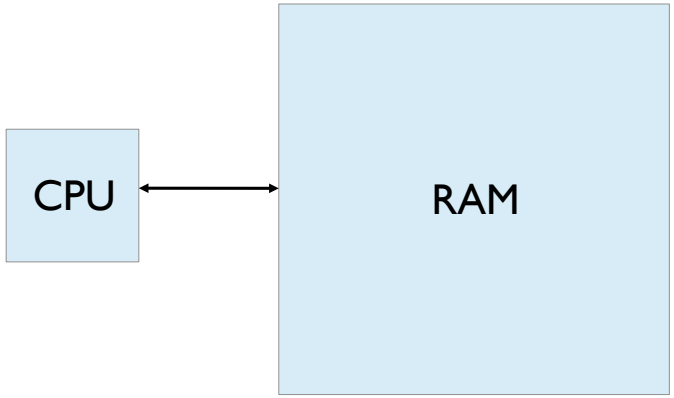
Thomas Rodgers
DRW Trading Group
trodgers@drw.com

Objectives

- Improve understanding of performance trade-offs inherent in modern hardware architectures
- How those tradeoffs impact data structure choices
- Make a case for preferring “modern” C++ constructs/idioms

Conceptual model

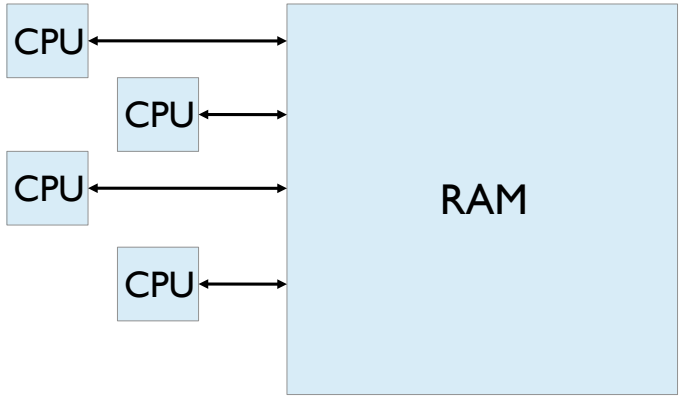


The architecture everybody would like to develop for, and usually does

Classic Von Neumann architecture

Or, because it’s all multicore these days, maybe this...

Conceptual Model



Last time this sort of simplistic model existed...

1979

Contemporary with the end of the era polyester shirts and disco
When this guy...

C with Classes

Started working on what would eventually become C++

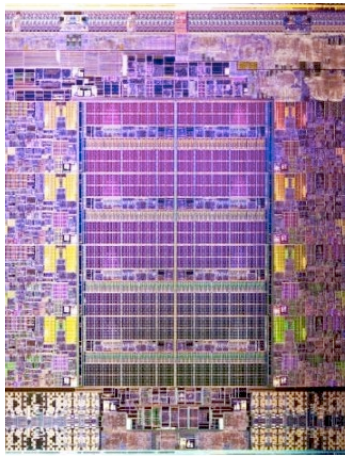
1998
C++
ISO standard

* Sandia National Labs' ASCI "Red", ~9200 PII's, peak numerical throughput ~1.3Tflops, first super computer to achieve a sustained TFLOP
* 850 kW, 1600 sq. ft. at a cost $55M
* Worlds fastest super computer until late 2000

C++03

* Back when we still thought these guys had a chance
* Opteron notable for defining what became the x86-64 ISA
* Fixed a number of bugs in the original C++98 standard, what most of us have worked with since

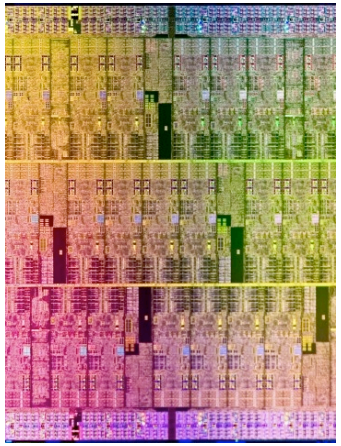
C++11



Most significant update to the language since 1998
CPU is an Intel Sandy Bridge 8C Xeon, ~2.7Bn transistors

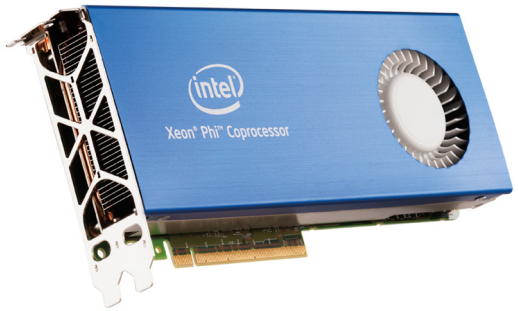
Today

≈



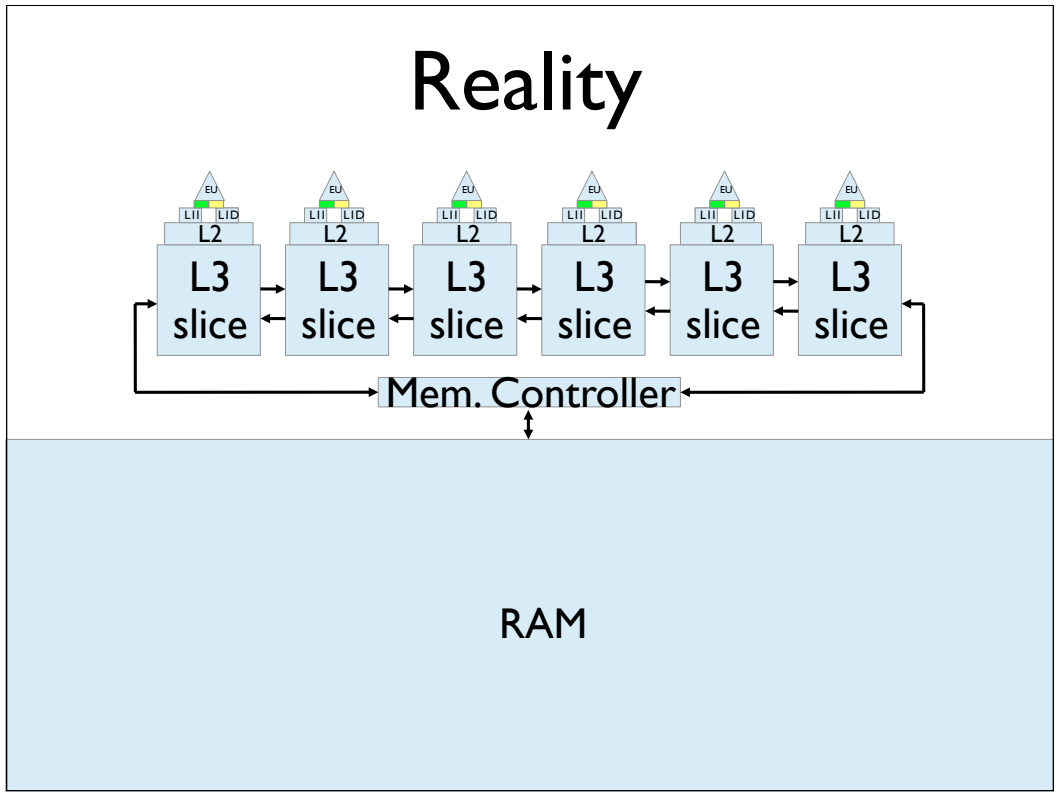
You can get roughly ASCI Red’s floating point performance on a chip

Today



as a $2500 add in card, draws about ~250 watts

Primary development tool chain, Intel C++ / Fortran



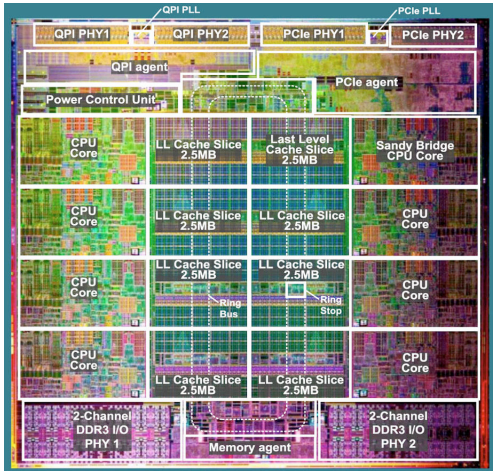
Reality looks more like this

Multiple cache tiers, with a very small, in relative terms, area of the CPU die dedicated to actually executing your code

The rest, by in large is there to hide memory latency

And, increasingly, control power distribution, integrate IO, memory control, etc.

Intel Xeon E5-2600



2.7Bn transistors
20MB L3 cache
8 Cores, each 256k L2 cache, 32k instruction + 32k data L1 cache
1.5k uop L0 cache

Size affects latency

- L1 cache, 32kb+32kb, ~4 clk
- L2 cache, 256kb, <12 clk
- L3 cache, 2.5mb/core, ~30 clk, unshared
- DRAM ~200clk, 60ns same socket

Big Memory != Fast Memory

L3 additional stats -
* 65 clk shared by another core/same socket
* 75 clk modified by another core/same socket
* 100-300 clk shared/modified by a core in a different socket

DRAM additional stats -
* 100ns different socket
* modern four issue super scalar CPU can execute 500-1000 instructions in the time it takes to load from DRAM

DRAM
Bandwidth vs Latency

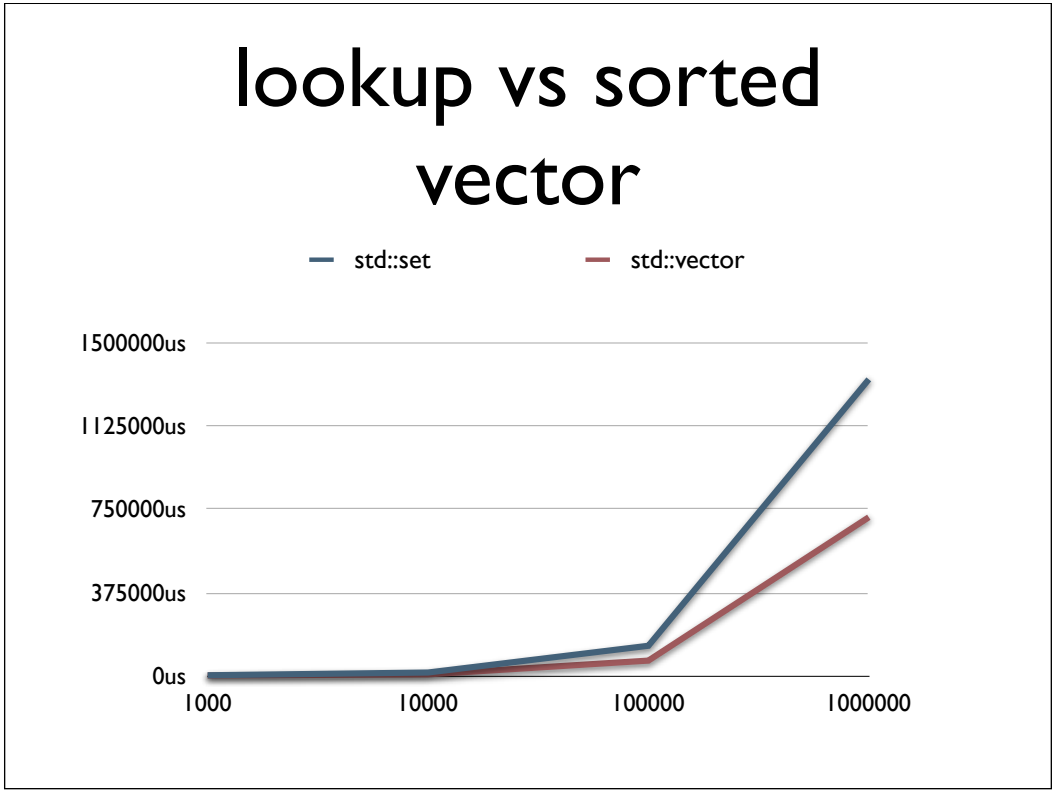
	1980	2012
Latency	225ns	60ns
Bandwith	13Mb/sec	13Gb/sec

Moore's law tends to benefit bandwidth more than latency
1000x improvement in bandwidth, 4x improvement in latency

STL set and map

- Typically implemented as a red/black tree
- Three pointers
  - left, right, parent
- Space for a key, or key/value pair
- 64 bit architecture
  - minimum size 32 bytes

For a map with string keys, minimum size is 72 bytes
Larger than a single cache line on x86-64



Lookups in an ordered vector are always faster, this has been the case for quite a while
Boost flat map/flat set give you a set/map interface to a sorted vector
Not a good choice where frequent insertions are required

"We assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus the bulk of the index must be kept on some backup store. The class of backup stores considered are pseudo random access devices which have a rather long access or wait time -- as opposed to a true random access device like core store -- and a rather high data rate once the transmission of physically sequential data has been initiated. Typical pseudo random access devices are: fixed and moving head discs, drums, and data cells."

- *Organization and maintenance of large ordered indexes*
Prof. Dr. R. Bayer, Dr. E. M. McCreight

In 1972 Rudolf Beyer and Ed McCraight published this paper on the B-tree data structure

Today it's used extensively for database indexes and increasingly file system organization

"We assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus the bulk of the index must be kept on some backup store. The class of backup stores considered are pseudo random access devices which have a rather long access or wait time -- as opposed to a true random access device like core store -- and a rather high data rate once the transmission of physically sequential data has been initiated. Typical pseudo random access devices are: fixed and moving head discs, drums, and data cells."

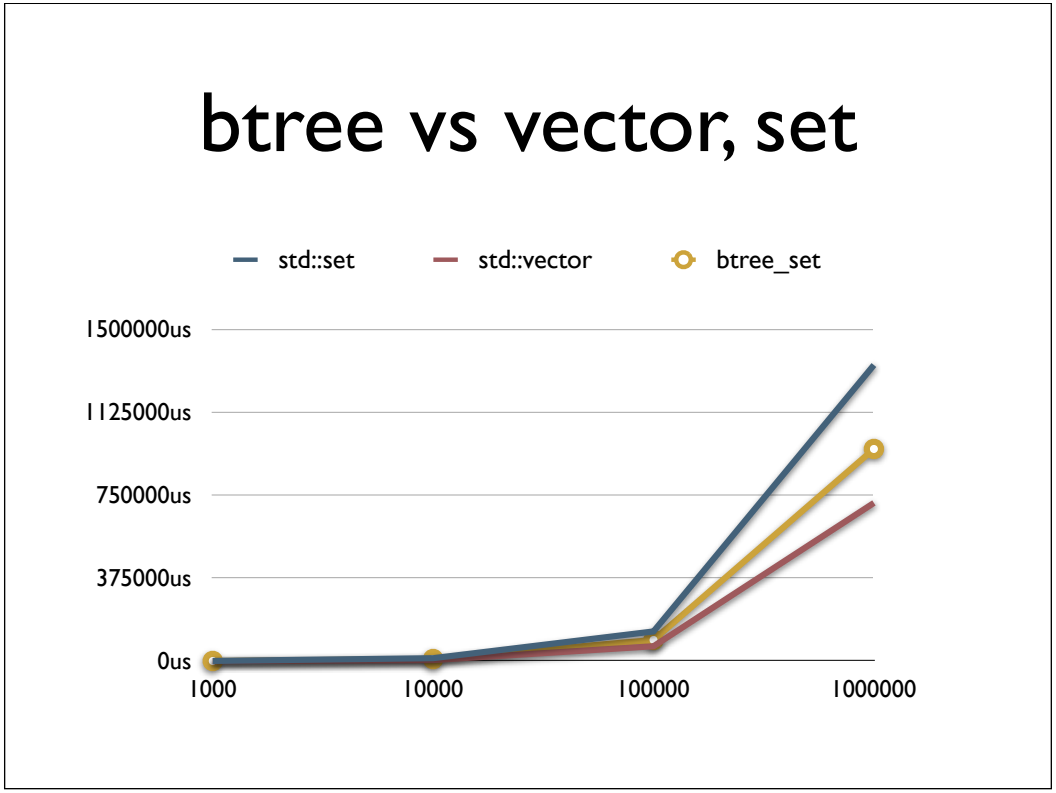
- *Organization and maintenance of large ordered indexes*
Prof. Dr. R. Bayer, Dr. E. M. McCreight

Sounds like a modern CPU cache

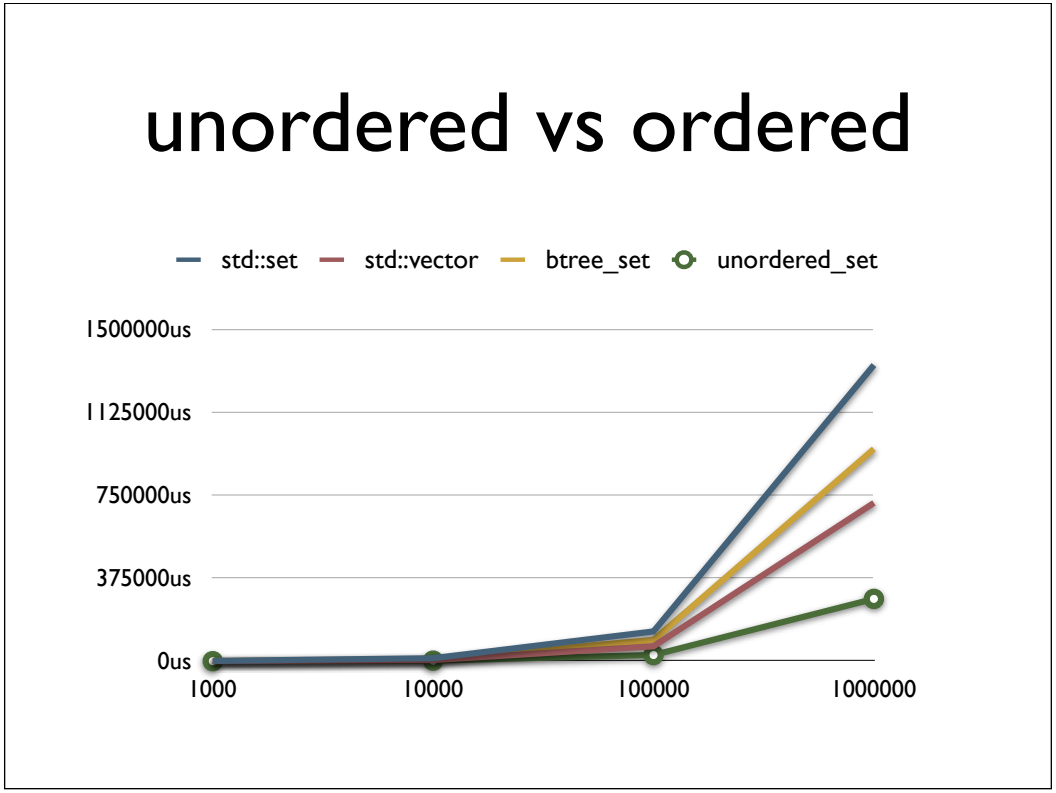
"We assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus the bulk of the index must be kept on some backup store. The class of backup stores considered are pseudo random access devices which have a rather long access or wait time -- as opposed to a true random access device like core store -- and a rather high data rate once the transmission of physically sequential data has been initiated. Typical pseudo random access devices are: fixed and moving head discs, drums, and data cells."

- *Organization and maintenance of large ordered indexes*
Prof. Dr. R. Bayer, Dr. E. M. McCreight

Sounds like a modern DRAM



Btree performance is substantially better, with much less overhead per key/value pair stored



Of course, if you only care about lookups...

Prefer compact data

- Prefer compact representations
- Prefer contiguous memory layouts
- Node based containers generally have poor locality
  - std::set, std::map, std::list

* or any sort of sparse data structure tend to perform poorly

Numbers to remember

- L1 Cache Reference - 0.5ns
- Branch mispredict - 5ns
- L2 Cache Reference - 7ns
- DRAM reference – 60-100ns
- Read 1MB sequentially from RAM - 250µs

C++11 Idioms

Prefer make_shared

Do this -

```
auto foo = std::make_shared<Foo>(a, b, c);
```

Rather than this -

```
std::shared_ptr<Foo> foo(new Foo(a, b, c));
```

First version makes a single allocation and placement-new's the contained type

No make_unique, yet, C++14

Prefer emplace

Do this -

```
std::vector<Foo> foos;

foos.emplace_back(a, b, c);
```

Rather than this -

```
std::vector<Foo> foos;

foos.push_back(Foo(a, b, c));
```

Where a container supports it

Avoids extra copy or move

Prefer making types

Do this -

```
struct point {
    float x;
    float y;
};

point upper, lower;
...
surface.draw_rect(upper, lower);
```

Not strictly a C++11 thing, but

Prefer making types

Rather than this -

```
float ux, uy, lx, ly;;
...
surface.draw_rect(ux, uy, lx, ly);
```

With a type, there's no possibility of confusing argument order
Compiler generates the same code

Small types by value

Do this -

```
struct point {
    float x;
    float y
};

void draw_rect(point upper, point lower) {
    ...
}
```

Small types by value

Do this -

```
struct point {
    float x;
    float y
};

void draw_rect(point const& upper, point const& lower) {
    ...
}
```

Compiler will tend to pass small types via registers, in this case upper and lower can both be enregistered
no possibility of aliasing with values, may end up being slightly faster

Prefer C++ to C

This

```
#include <cstdlib>

int compare_ints(const void* a, const void* b) {
    int* arg1 = (int*) a;
    int* arg2 = (int*) b;
    if (*arg1 > *arg2) return -1;
    else if (*arg1 == *arg2) return 0;
    else return 1;
}

...

qsort(a, size, sizeof(int), compare_ints);
```

Also not strictly a C++11 thing, but if you are new to C++ or in the habit of using C++ as a “better” C

Prefer C++ to C

Is much slower than this

```
std::sort(s.begin(), s.end(), std::greater<int>());
```

about 2.5x slower
qsort is part of the C standard library, does things the C way, throws away all type information, no opportunity to inline comparison function
Same idea goes for copy vs. memcpy

std::sort is much more succinct

Prefer STL algorithms

Do this -

```
vector<position> positions;
...
vector<position> expired;
vector<position> unexpired;
partition_copy(begin(positions), end(positions),
                        inserter(expired, end(expired)),
                        inserter(unexpired, end(unexpired)),
                        is_expired);
```

The abstraction is free, generates the same code as if you had hand written it

Prefer STL algorithms

Instead of this -

```
vector<position> positions;
...
vector<position> expired;
vector<position> unexpired;
for (auto it = begin(positions); it != end(positions); ++it) {
    if (is_expired(*it))
        expired.emplace_back(*it);
    else
        unexpired.emplace_back(*it);
}
```

Prefer STL algorithms

Or even this -

```
vector<position> positions;
...
vector<position> expired;
vector<position> unexpired;
for (auto p : positions) {
    if (is_expired(p))
        expired.emplace_back(p);
    else
        unexpired.emplace_back(p);
}
```

Prior to C++11 there was an argument for not using STL style algorithms, the syntax was clumsy if the default predicate wasn't sufficient, C++11 lambda syntax greatly improves matters, and generic lambdas in C++14 make it cleaner still.

Algorithms state up front, what they are going to do, e.g. for_each, you know when reading code that it will visit each element in the range, a naked for loop, you have to consider at least four things, init, condition, increment, body

Prefer STL algorithms

- Parallelized and vectorized abstractions
  - Standards proposal N3354

Likely coming in some form, probably C++17
If you are in the habit of expressing your code in terms of operations on ranges, using things like transforms, it will be a fairly direct process to enable parallel or vectorized versions of your code

To some extent you can already do this using Thrust

Thrust

http://thrust.github.com

- Modeled on the STL
- Host and device vectors
  - Similar to std::vector
  - Handle details of transfers to/from device memory

Thrust

http://thrust.github.com

- Algorithms expressed as functors which transform iterator ranges
- Also supports "fusing" transformations into single device calls via *fancy iterators*
  - transform_iterator lazily applies a functor to an underlying range to generate new values

Thrust

http://thrust.github.com

- Backends Target
  - CUDA - nVidia GPGPUs
  - OpenMP - clusters of servers
  - Intel's TBB - multiple cores, same machine

Seems likely it will also be able to target the Xeon Phi co-processors I mentioned earlier
as that uses thread building blocks to express concurrent operations

Thank you, questions?