BLUESHIFT MEMORY

Introducing the Cambridge Architecture

The evolution of the Stored Program Machine

BLUESHIFT MEMORY

Problem statement

Modern computing workloads

- exhibit high memory intensity
- are becoming progressively structured
- when data sets grow,

caches become less effective

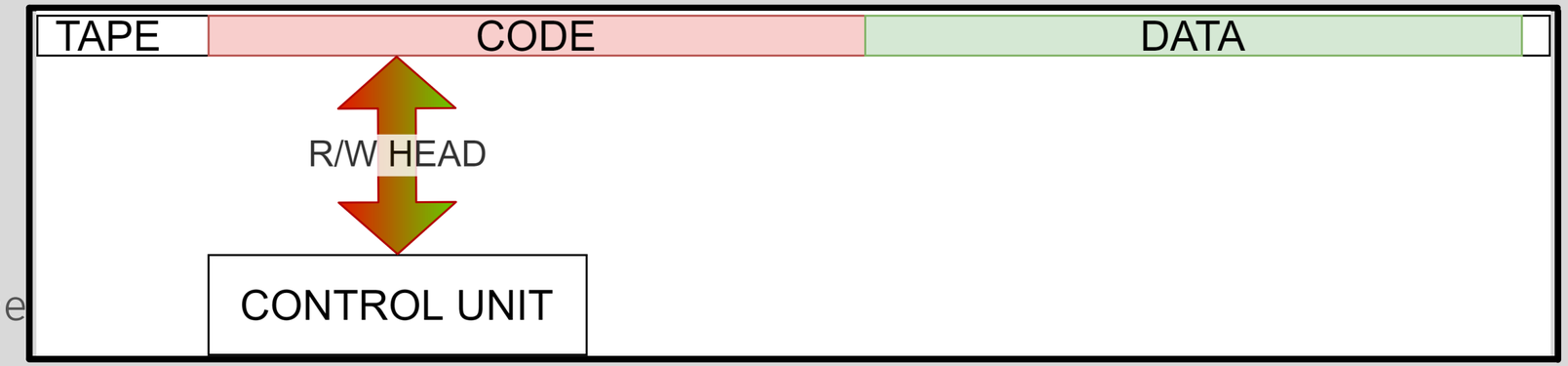
- memory access is becoming

dominant energy consumer

Generation 0
Turing Machine

ABSTRACTION
Computing is code and data - both could be stored

ADVANTAGE
Creation of stored program machine
The base of Computing



1936

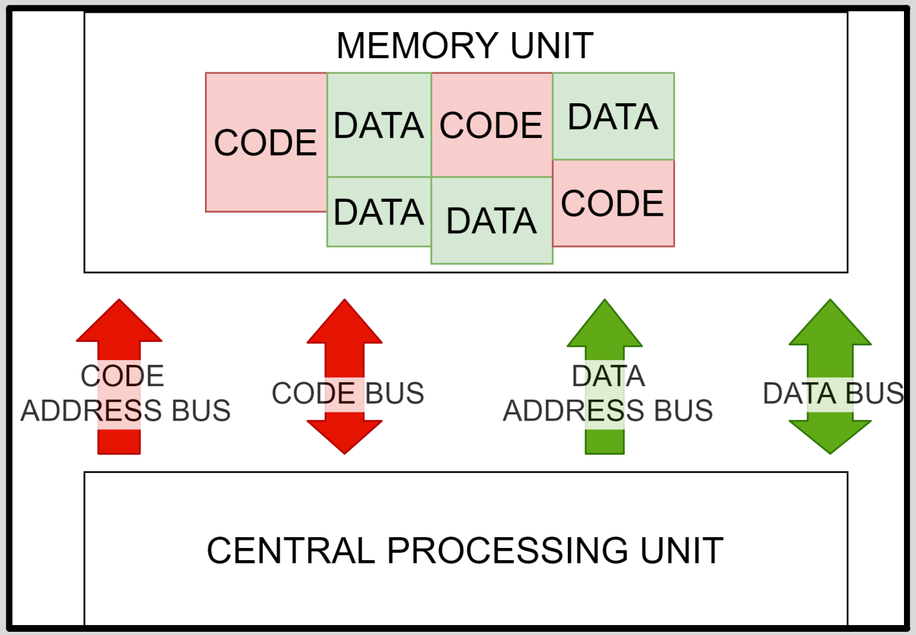
Generation 1
Harvard Architecture

ABSTRACTION

Code and Data separation
RAM Machine

ADVANTAGE

Different infrastructure for code and data handling improved performance



1944

Generation 1
von Neumann/Princeton architecture

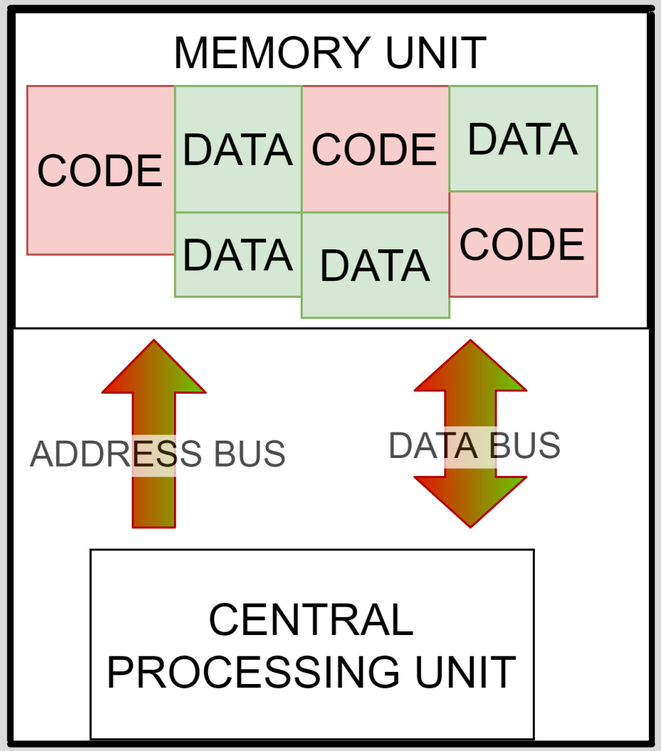
ABSTRACTION

CPU usage
Code/Data stored in Random Access Memory
RAM uses addresses to store/access data

ADVANTAGE

Shared infrastructure for code and data reduced costs
Address manipulation (pointers)
Birth of data structures

1945



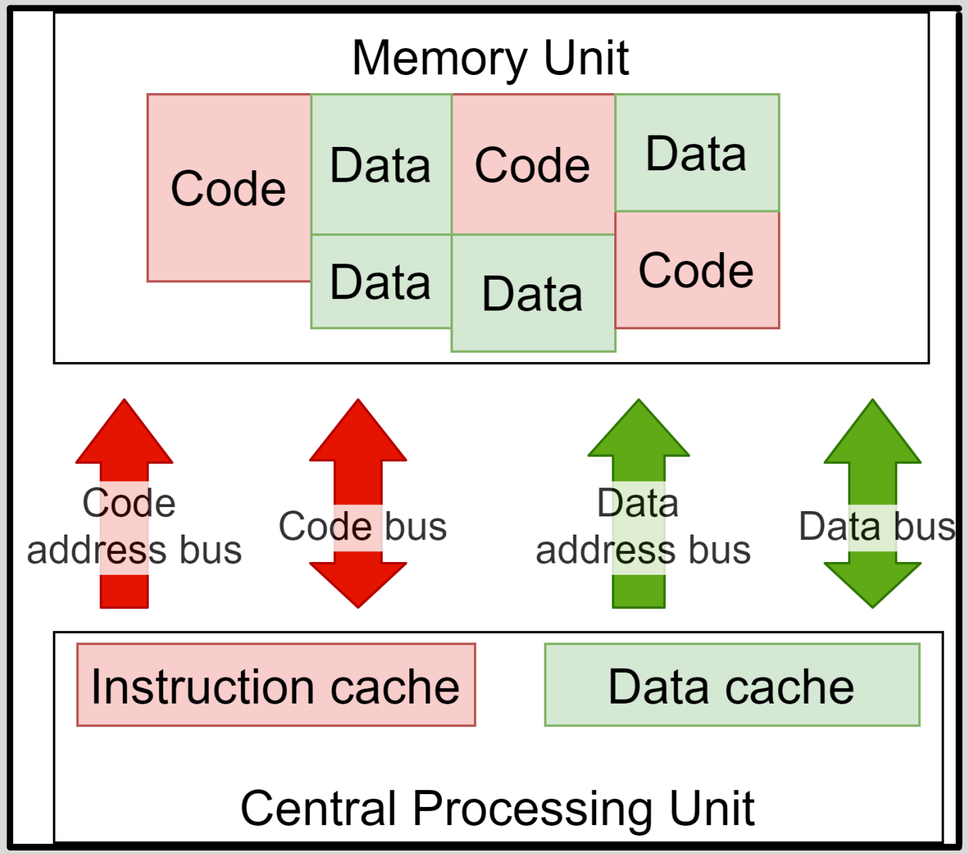
Generation 2
Cache systems

ABSTRACTION
Working sets for code and data

ADVANTAGE

Lower average latency of memory accesses
Improved cycle time/performance
Better resource utilization on external I/O

1976



Back to our Problem statement

Modern computing workloads

- exhibit high memory intensity
- are becoming progressively structured
- when data sets grow,

caches become less effective

- memory access is becoming

dominant energy consumer

Foreach and map-reduce

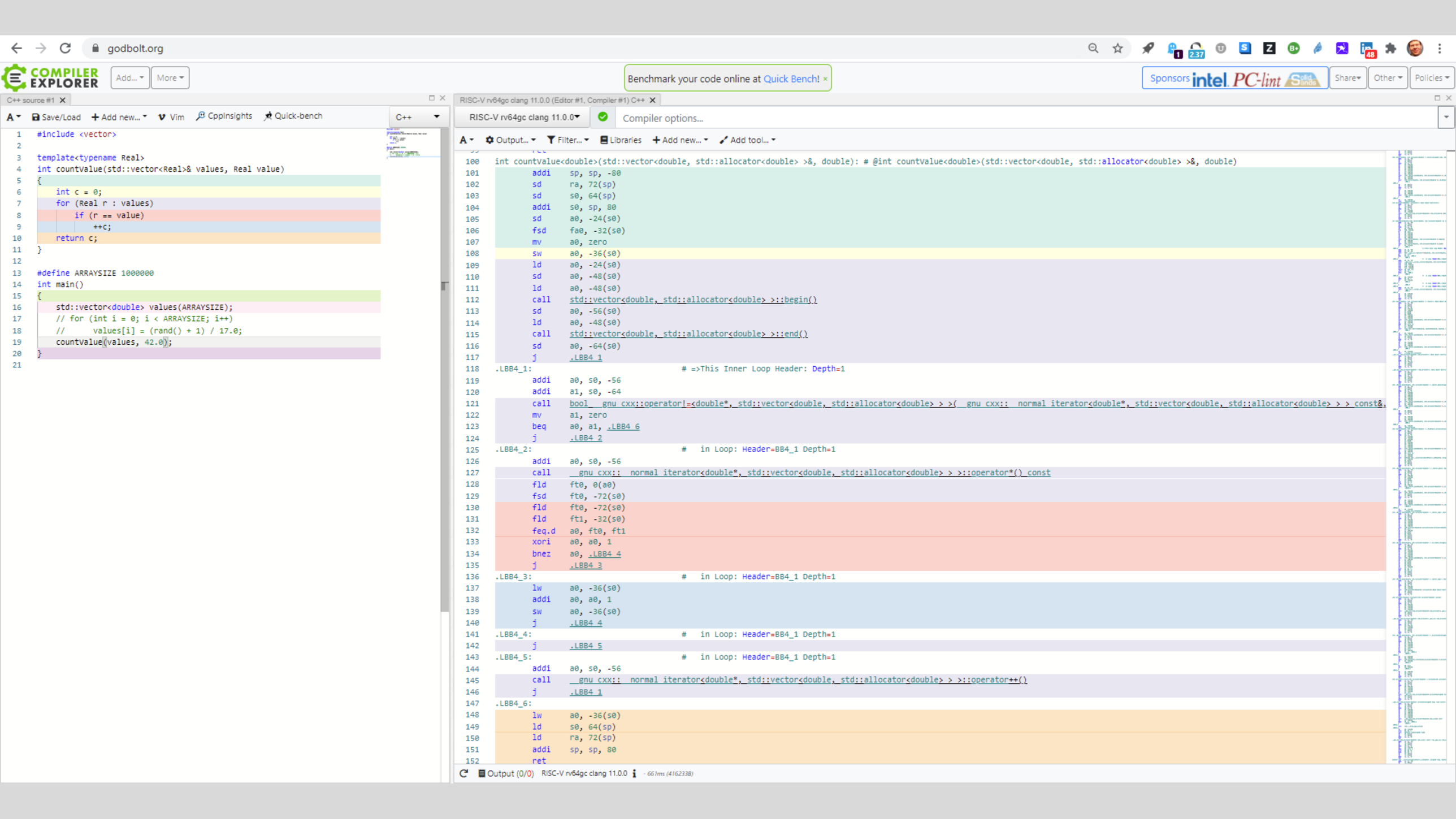
```java
    public static void main(String[] args) {

        List<String> nameList = new ArrayList<String>();

        nameList.add("Java");
        nameList.add("Kotlin");
        nameList.add("Android");

        // only single statement to be executed for each i
        nameList.forEach(name -> System.out.println(name)
    }
}
```

```cpp
template<typename Real>
int countValue(const std::vector<Real>& V, const Real& ref) {
    int c = 0;
    for (auto v: V) {
        if (v == ref) ++c;
    }
    return c;
}
```

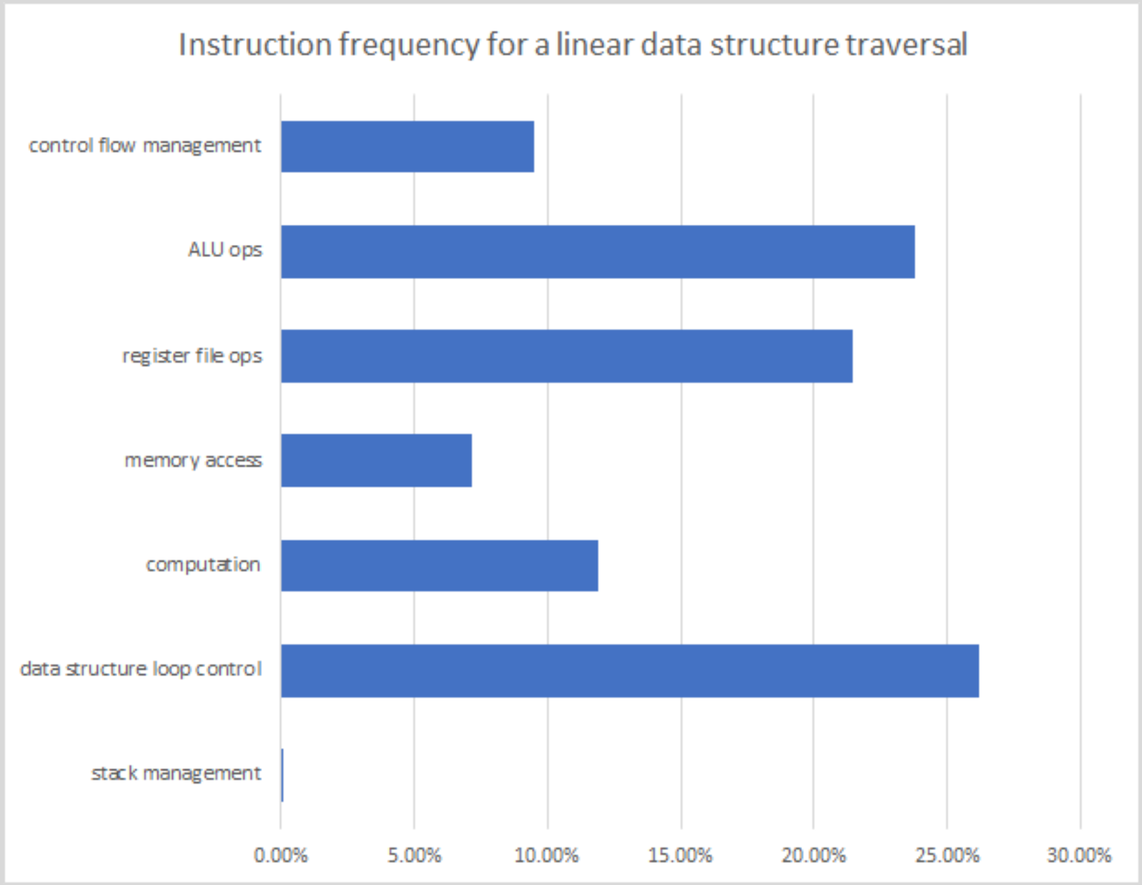
```go
var numbers = map[int]string{1: "one", 2: "two", 3: "three"}

func countValue(numbers map[int]string, value string) (count int) {
    for _, v := range numbers {
        if value == v {
            count++
        }
    }
    return
}
```

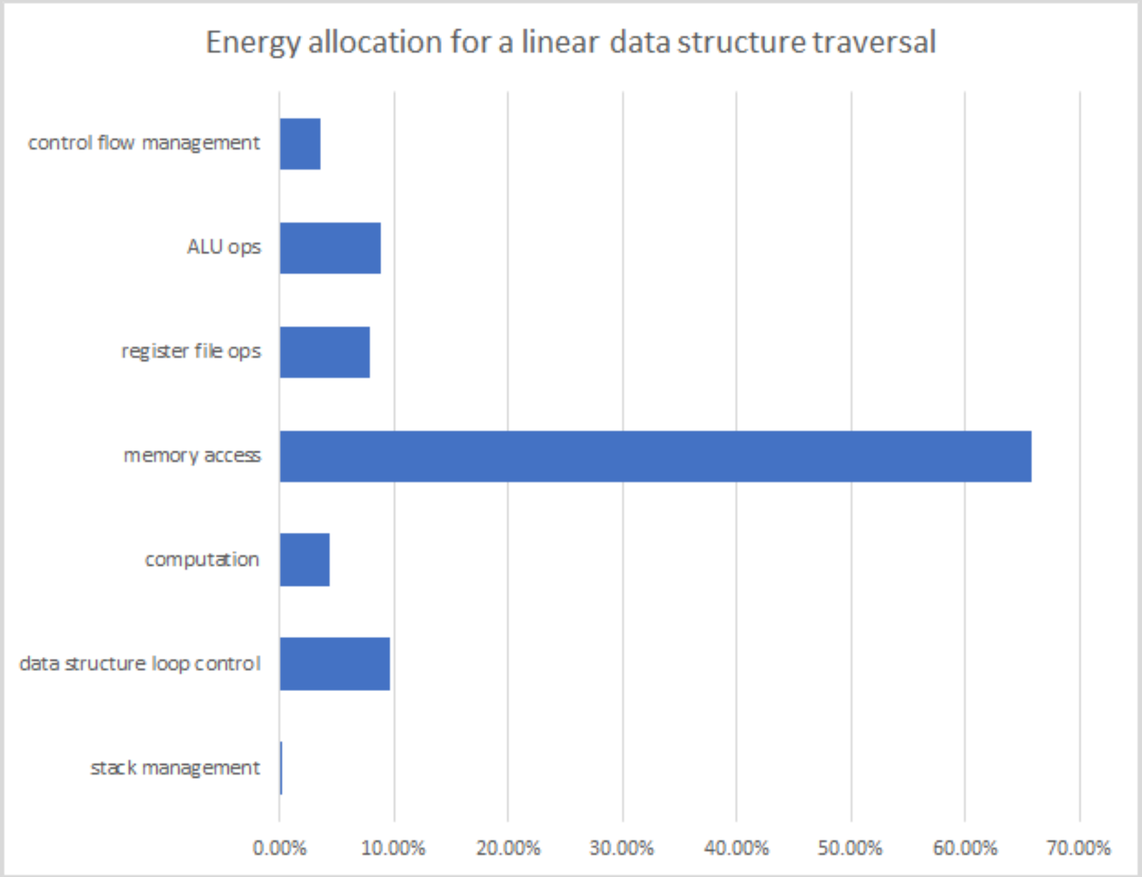
```python
names = ['tom', 'john', 'simon']

for element in names:
    operate(element)
```

```javascript
const companies = ["Apple", "Google", "Facebook"];

companies.forEach(company => {
    console.log(company);
});
```


godbolt.org
COMPILER EXPLORER
Add...
More
Benchmark your code online at Quick Bench!
Sponsors intel. PC-lint
Share
Other
Policies
C++ source #1
Save/Load
Add new...
Vim
Cppinsights
Quick-bench
C++
#include <vector>
template<typename Real>
int countValue(std::vector<Real>& values, Real value)
{
int c = 0;
for (Real r : values)
if (r == value)
++c;
return c;
}
#define ARRAYSIZE 1000000
int main()
{
std::vector<double> values(ARRAYSIZE);
// for (int i = 0; i < ARRAYSIZE; i++)
// values[i] = (rand() + 1) / 17.0;
countValue(values, 42.0);
}
RISC-V rv64gc clang 11.0.0 (Editor #1, Compiler #1) C++
RISC-V rv64gc clang 11.0.0
Compiler options...
Output...
Filter...
Libraries
Add new...
Add tool...
int countValue<double>(std::vector<double, std::allocator<double> >&, double): # @int countValue<double>(std::vector<double, std::allocator<double> >&, double)
addi sp, sp, -80
sd ra, 72(sp)
sd s0, 64(sp)
addi s0, sp, 80
sd a0, -24(s0)
fsd fa0, -32(s0)
mv a0, zero
sw a0, -36(s0)
ld a0, -24(s0)
sd a0, -48(s0)
ld a0, -48(s0)
call std::vector<double, std::allocator<double> >::begin()
sd a0, -56(s0)
ld a0, -48(s0)
call std::vector<double, std::allocator<double> >::end()
sd a0, -64(s0)
j .LBB4_1
.LBB4_1: # =>This Inner Loop Header: Depth=1
addi a0, s0, -56
addi a1, s0, -64
call bool __gnu_cxx::operator!=<double*, std::vector<double, std::allocator<double> > >(__gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > > const&,
mv a1, zero
beq a0, a1, .LBB4_6
j .LBB4_2
.LBB4_2: # in Loop: Header=BB4_1 Depth=1
addi a0, s0, -56
call __gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > >::operator*() const
fld ft0, 0(a0)
fsd ft0, -72(s0)
fld ft0, -72(s0)
fld ft1, -32(s0)
feq.d a0, ft0, ft1
xori a0, a0, 1
bnez a0, .LBB4_4
j .LBB4_3
.LBB4_3: # in Loop: Header=BB4_1 Depth=1
lw a0, -36(s0)
addi a0, a0, 1
sw a0, -36(s0)
j .LBB4_4
.LBB4_4: # in Loop: Header=BB4_1 Depth=1
j .LBB4_5
.LBB4_5: # in Loop: Header=BB4_1 Depth=1
addi a0, s0, -56
call __gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> > >::operator++()
j .LBB4_1
.LBB4_6:
lw a0, -36(s0)
ld s0, 64(sp)
ld ra, 72(sp)
addi sp, sp, 80
ret
Output (0/0)
RISC-V rv64gc clang 11.0.0
661ms (4162338)

Big Data/Streaming workloads





Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion*	Deletion**	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Graph Operations						
Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query*
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

Heap Operations						
Type	Time Complexity					
	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)	O(log n)	O(log n)	O(m+n)
Binomial Heap	O(1)	O(log n)	O(log n)	O(1)	O(log n)	O(log n)
Fibonacci Heap	O(1)	O(log n)	O(1)	O(1)	O(log n)	O(1)

Data structure management has become a service.

Custom hardware can dramatically improve energy efficiency.

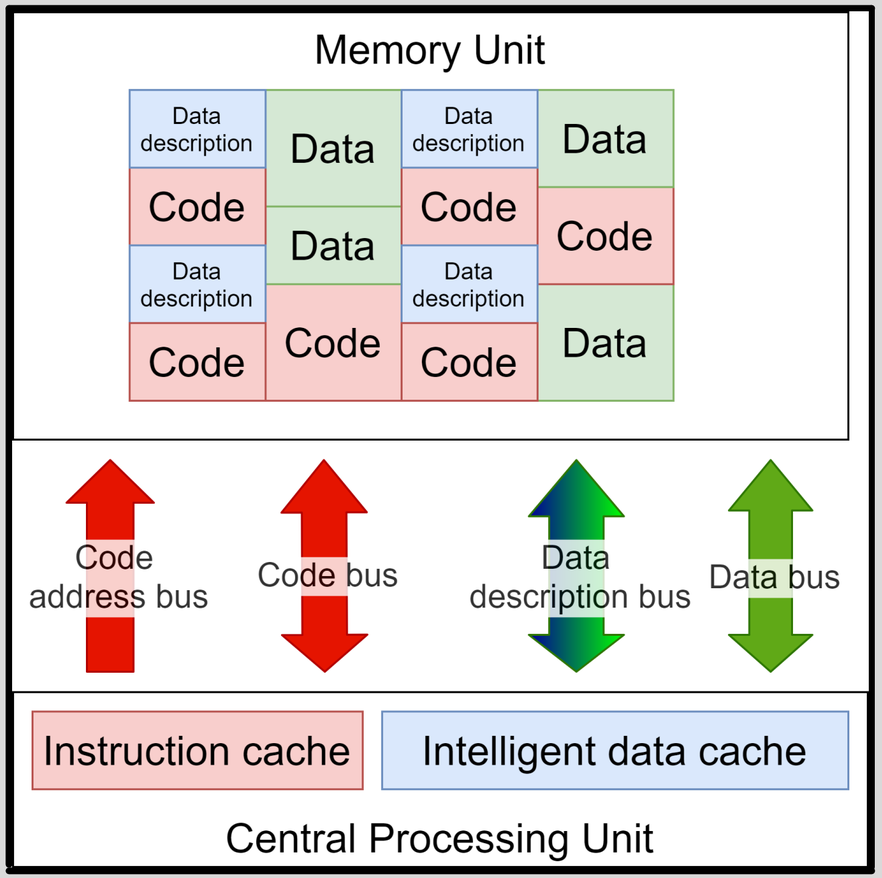
Cambridge Architecture

makes memory responsible for data structure management

Next Generation Cambridge Architecture

ABSTRACTION

Most memory accesses are not random
Memory becomes aware of data structure
Data structure traversal and scheduling are moved to memory
Memory becomes an active data structure management service



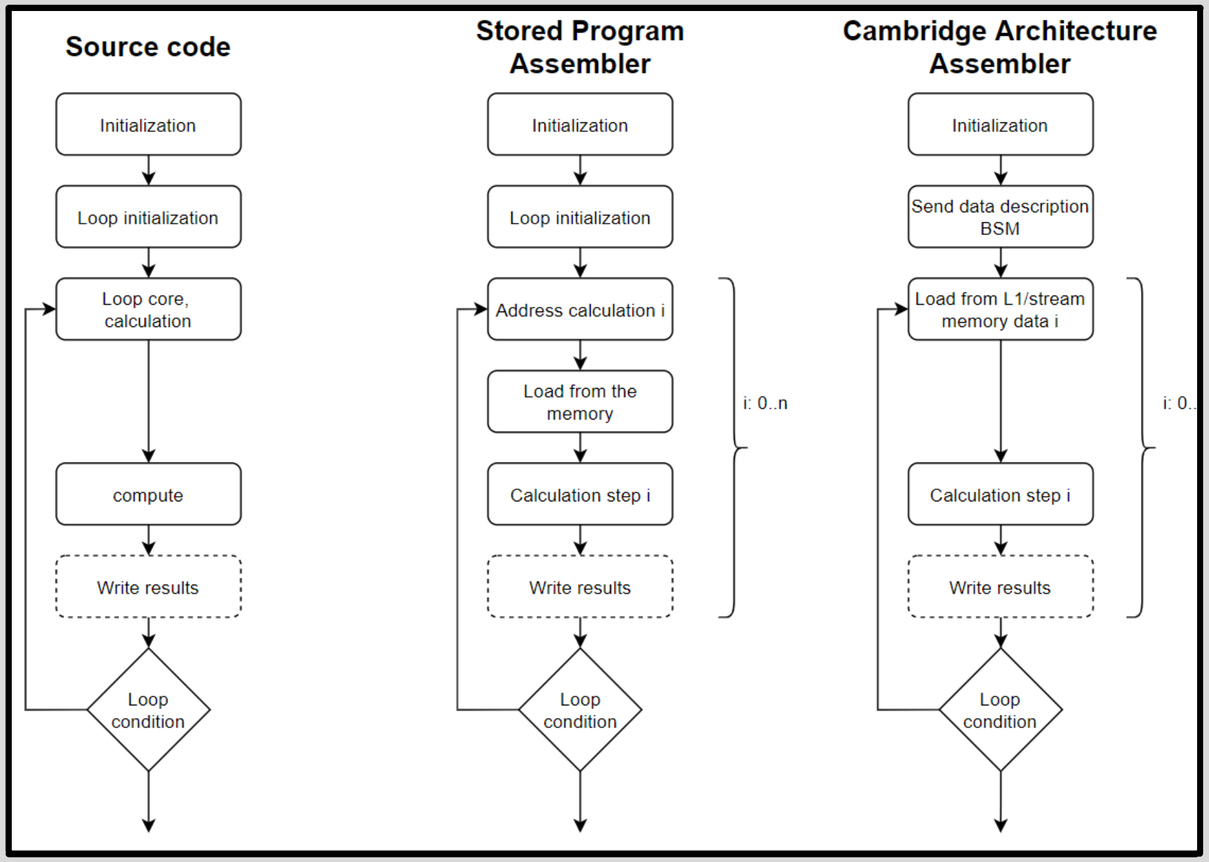
2017

Cambridge Architecture

OPERATION

Data structure traversal and schedule
owned by memory
transformed into a stream

Processor execution transition
from request/reply to data-driven
stream bypasses caches



Cambridge Architecture BlueShift Memory streaming

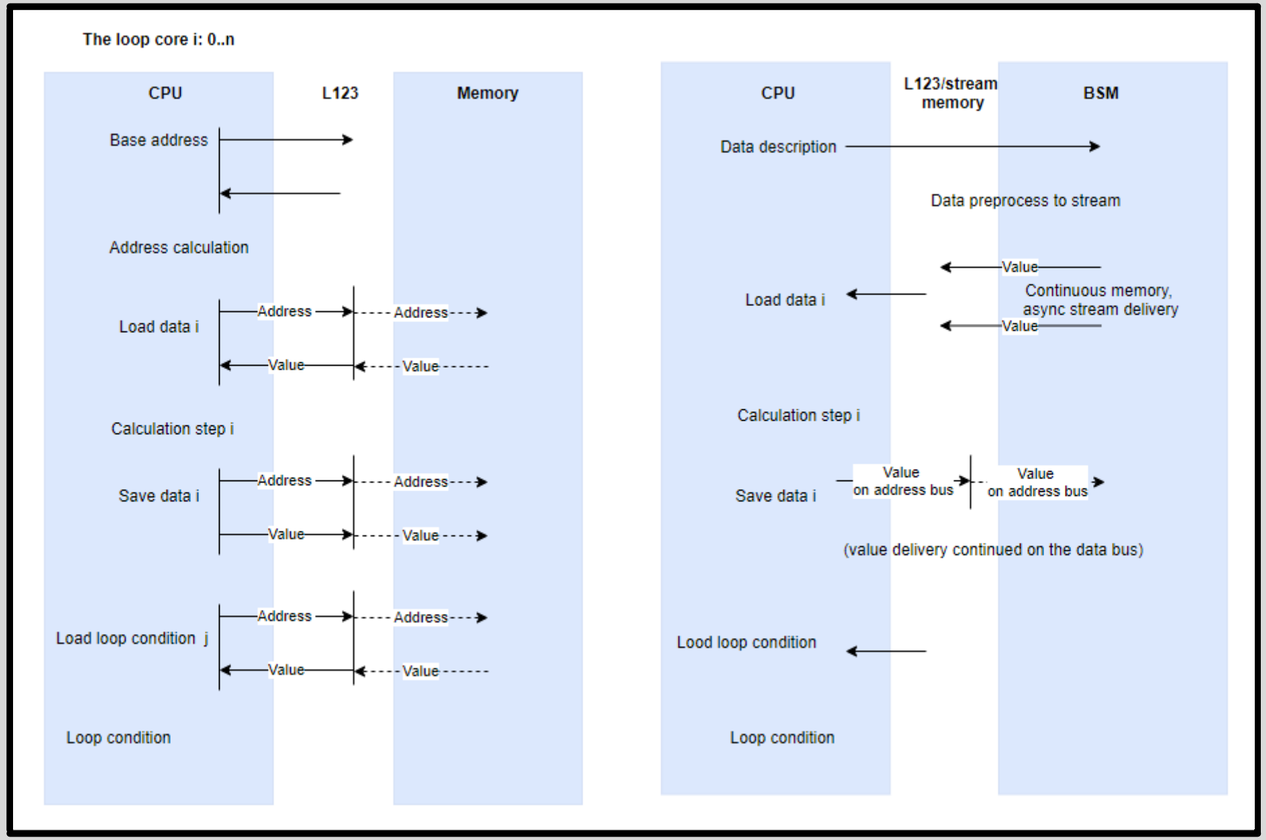
OPERATION

- BlueShift Memory receives data descriptor

- Preprocesses the schedule

- Transform data access into streams
  Send to CPU

- Receive update stream
  Update data in memory



BLUESHIFT MEMORY

Cambridge Architecture

benefits

Cambridge Architecture benefits

Zero-latency memory

media workloads
big data operators

Energy Efficiency Improvements

50%+

Performance Improvements

2-1000x

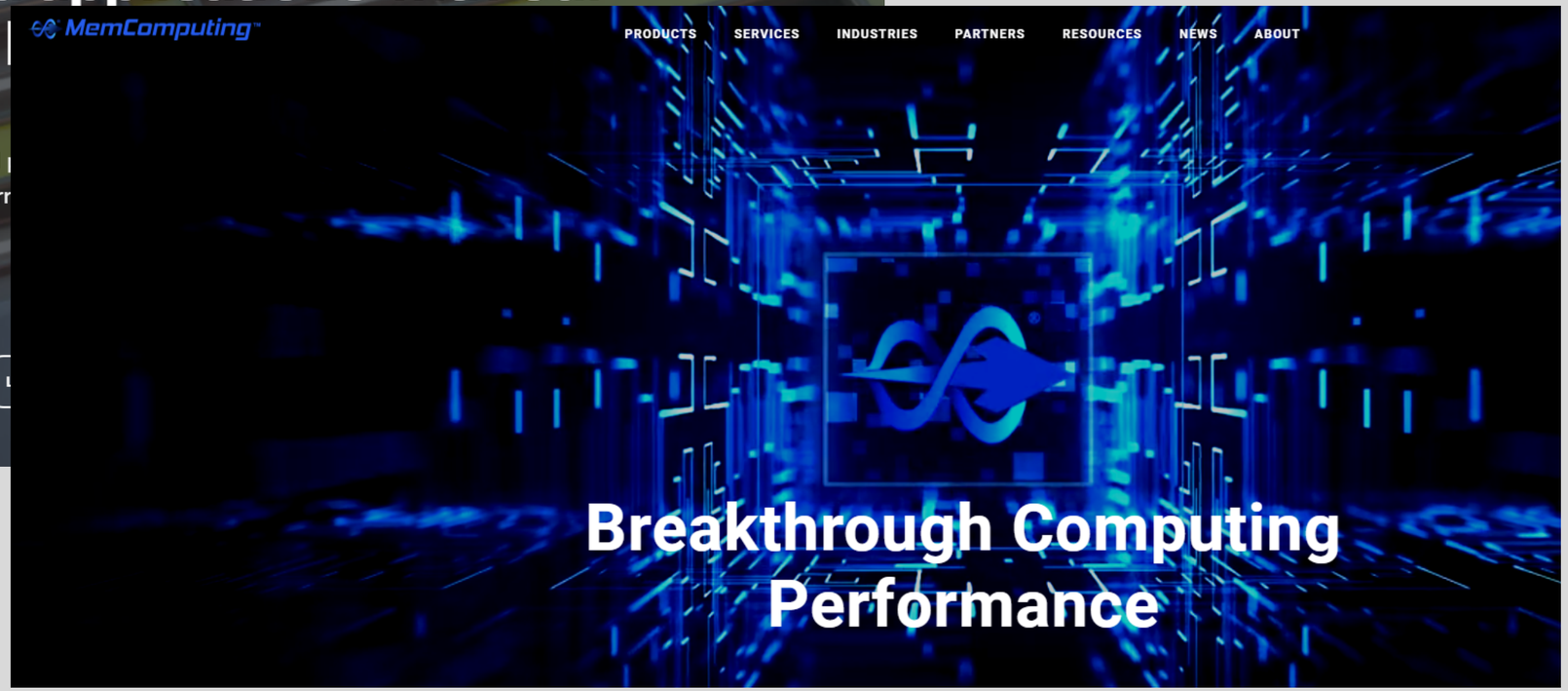
ACM DIGITAL LIBRARY

☰ Article Navigation

Exposing Memory Access Patterns to Improve Instruction and Memory Efficiency in GPUs

NEAL C. CRAGO, MARK STEPHENSON, and STEPHEN W. KECKLER, NVIDIA

ACM Trans. Archit. Code Optim., Vol. 15, No. 4, Article 45, Publication date: October 2018.
DOI: https://doi.org/10.1145/3280851



Demonstration Prototype

Youtube videos

Sorting

https://www.youtube.com/watch?v=figcOHfk0Js

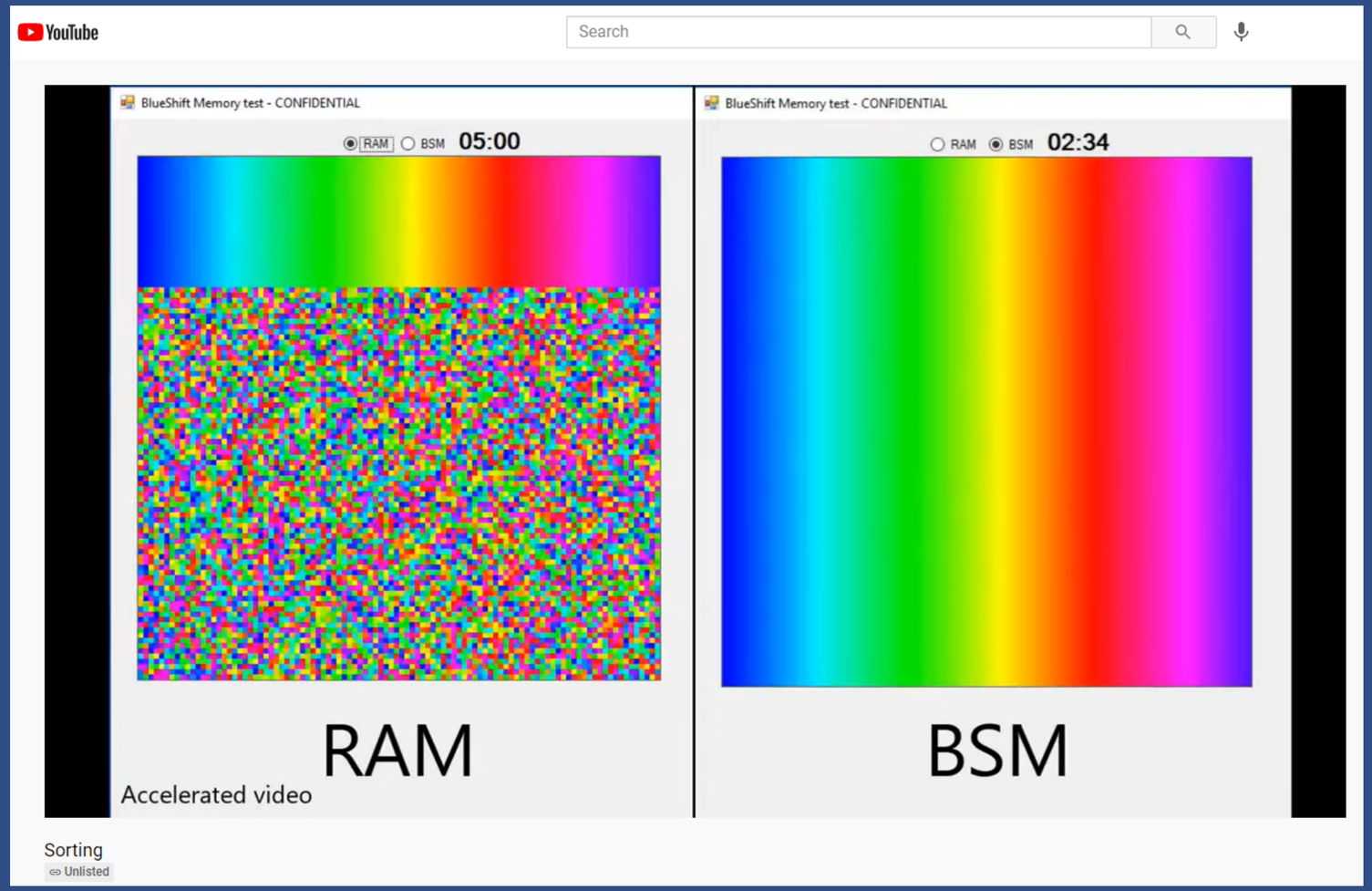
Image Processing

https://www.youtube.com/watch?v=hS0MwG8hDDM

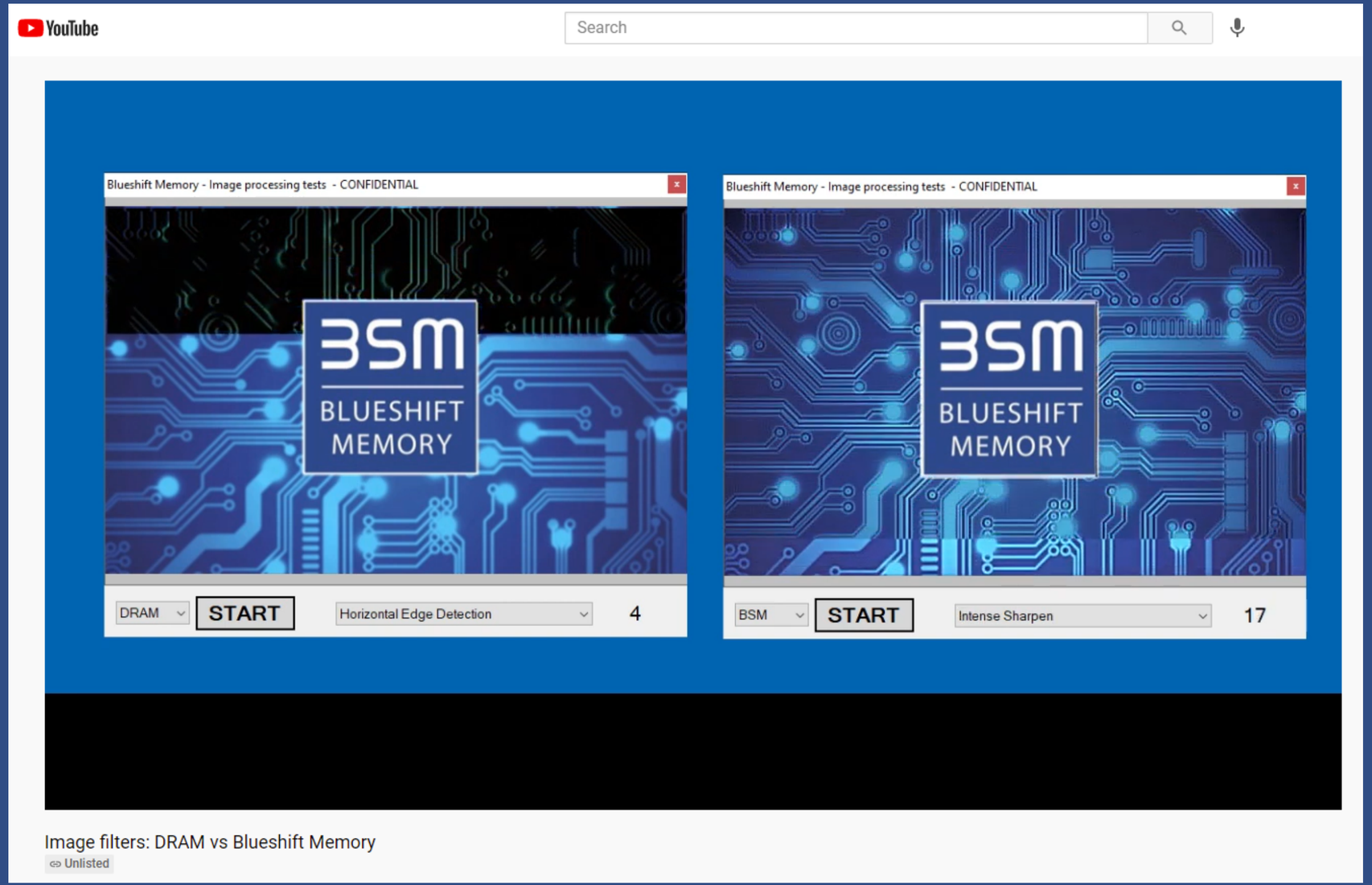
Computer Vision

https://www.youtube.com/watch?v=fT-s-BppSFY

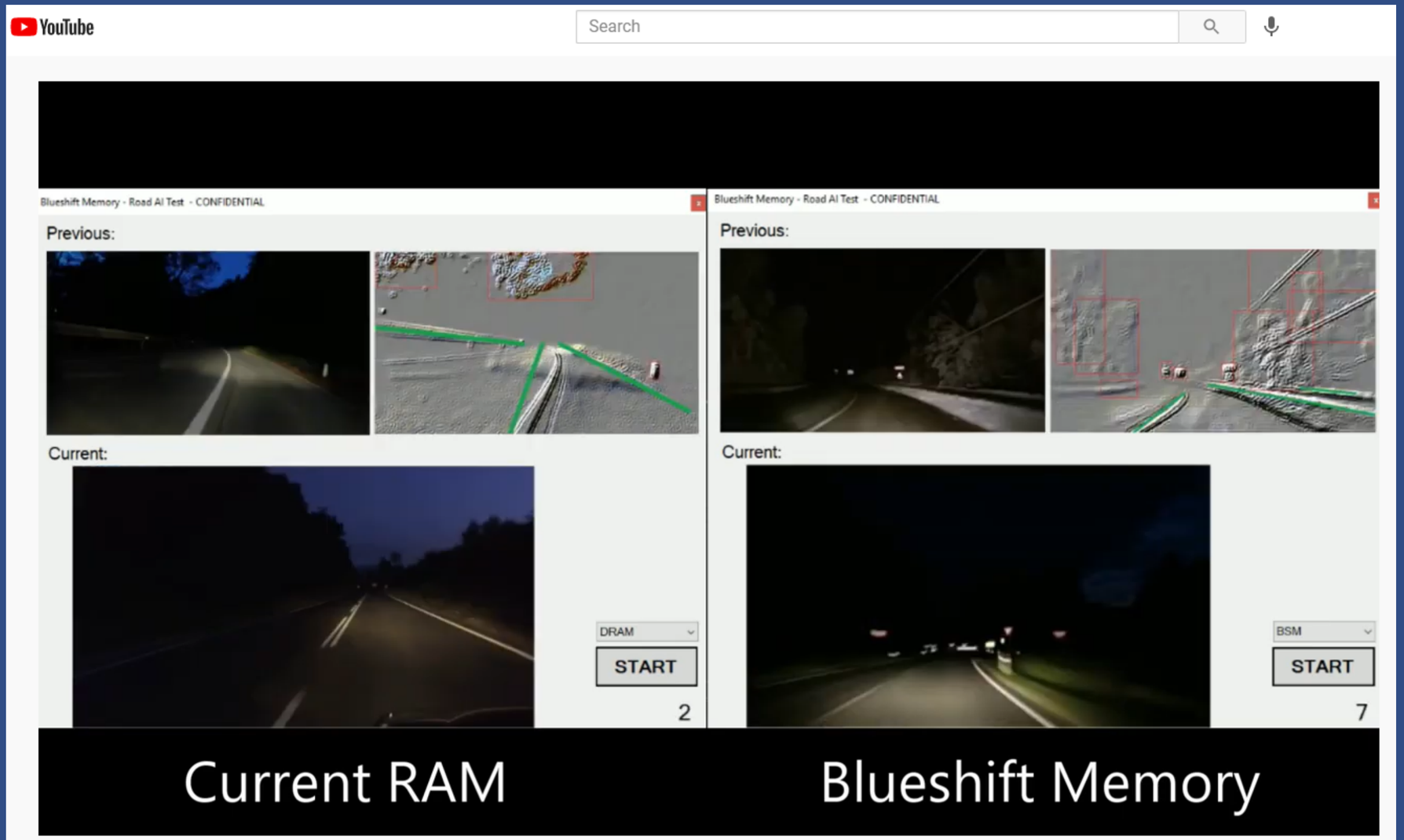
BSM speed-up 10x



BSM speed-up 2x



BSM speed-up 4x



AI data delivery test

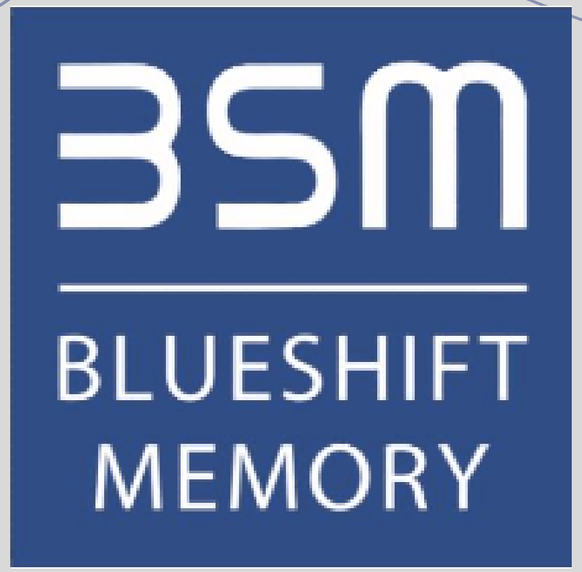
Unlisted

Conclusion

BLUESHIFT MEMORY

Modern big data and media workloads benefit from active memory

Cambridge Architecture enables performance and energy efficiency improvements as a natural evolution



info@blueshiftmemory.com