

What happens when a
computer tries to remember too
much?

<https://tinyurl.com/not404actually>

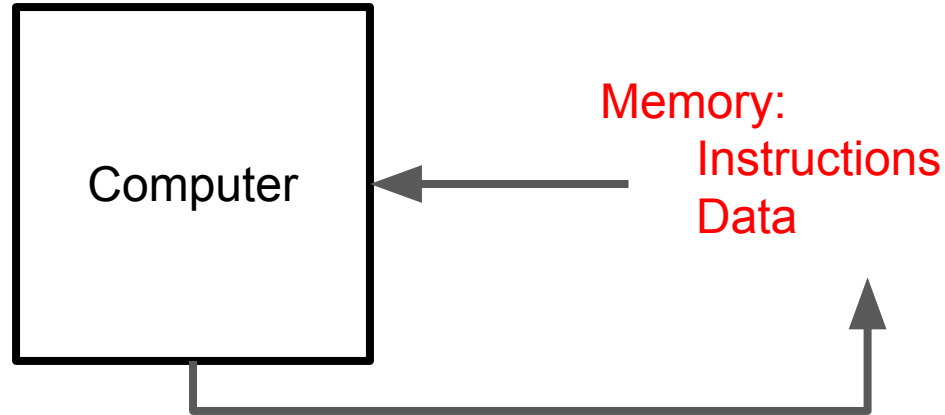
Presentation outline

- What is a computer, actually.
- The widely spread memory model.
- The nerdier memory model*.
- Ok let's hack* Google!!1!

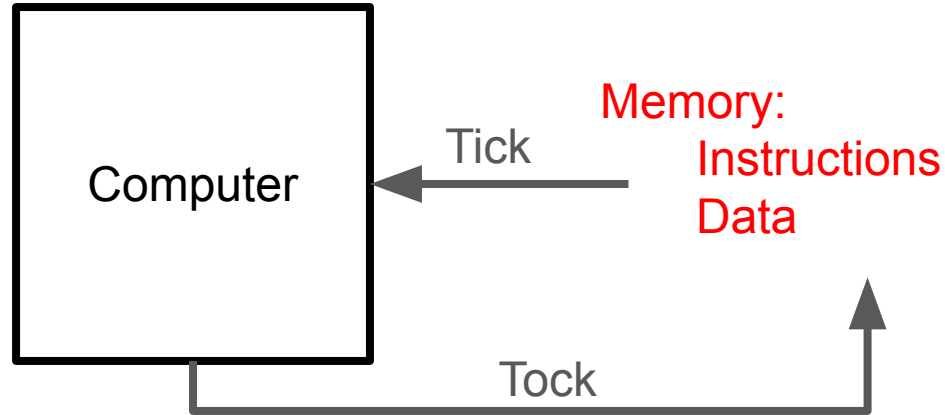
Presentation outline

- What is a computer, actually.
- The widely spread memory model.
- The nerdier memory model*.
- Ok let's hack* Google!!1!

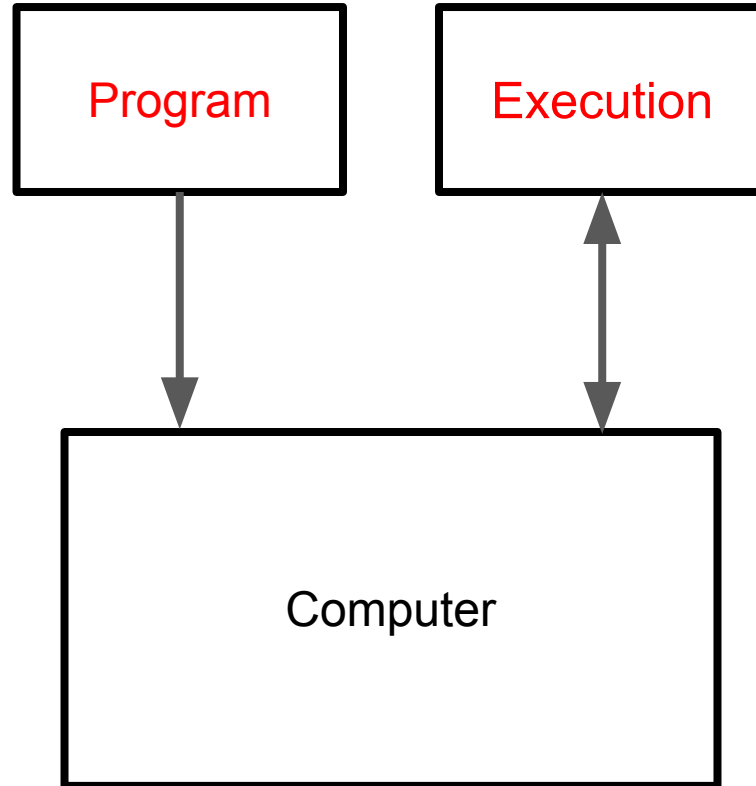
Computers execute binary instructions from memory,
one thing at a time



Computers execute binary instructions from memory,
one thing at a time

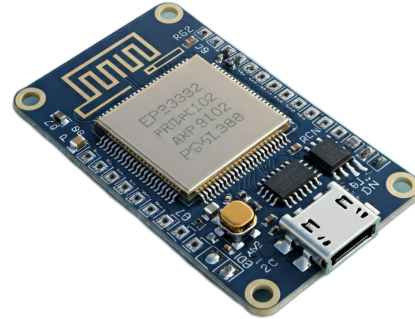
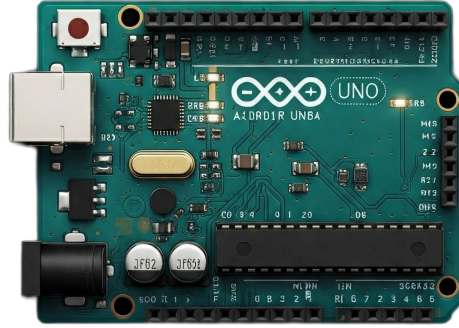


Harvard architecture separates “program” from “execution”

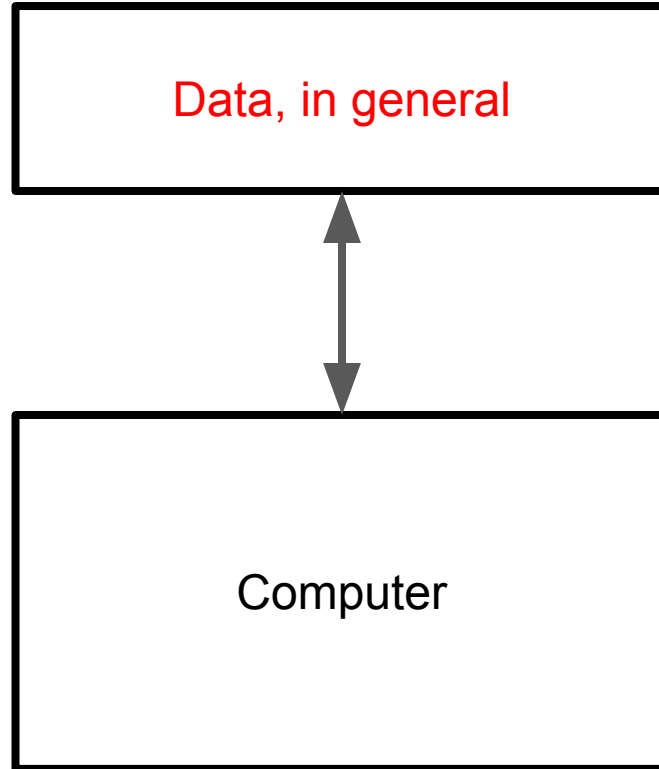


Harvard architecture is common among microcontrollers

- Arduino
- ESP32



Princeton architecture groups everything



Princeton* architecture is the norm

- Laptops
- Phones
- RaspberryPi's
- Servers



Presentation outline

- What is a computer, actually.
- The widely spread memory model.
- The nerdier memory model*.
- Ok let's hack* Google!!1!

Latency is ETA, bandwidth is number of traffic lanes

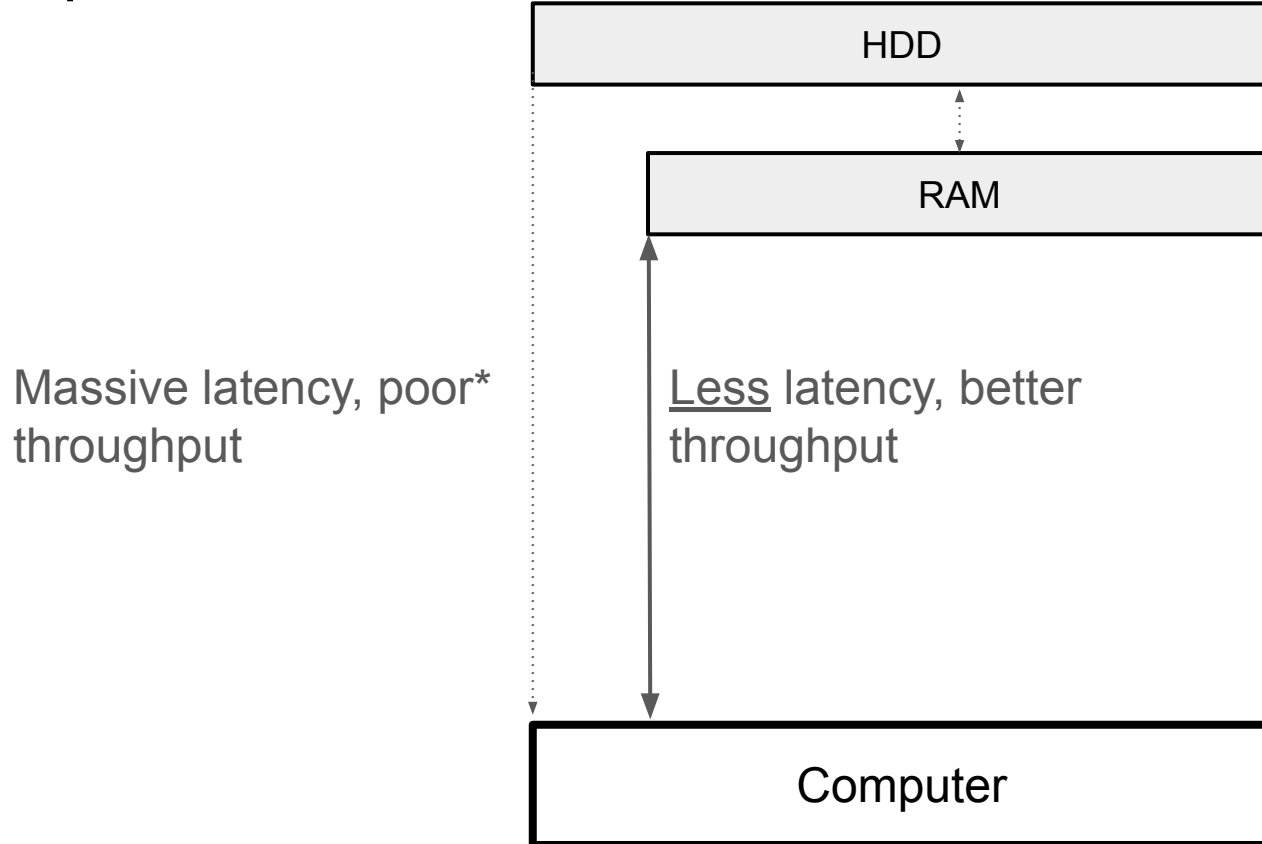
- Latency of memory is the time it takes between the request and data arrival



- Bandwidth of memory is how much data can be moved per unit time



The computer lives in RAM and interacts with HDD



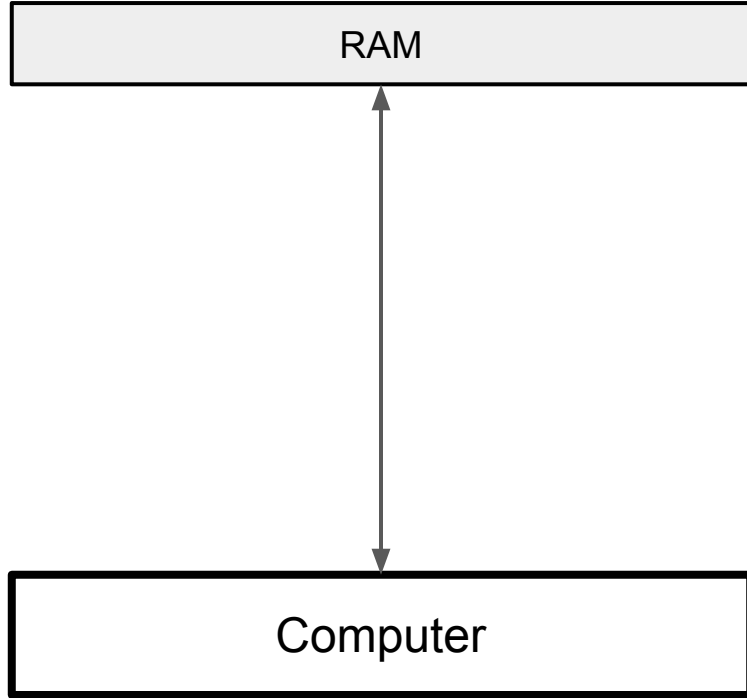
HDD is permanent and RAM is volatile

- RAM is much smaller than HDD
- HDD can be disk or solid state
 - Drastically different latency and random access performance
- Computers can use HDD when they run out of RAM (swap)
- Data from HDD can be cached in RAM (superfetch)

Presentation outline

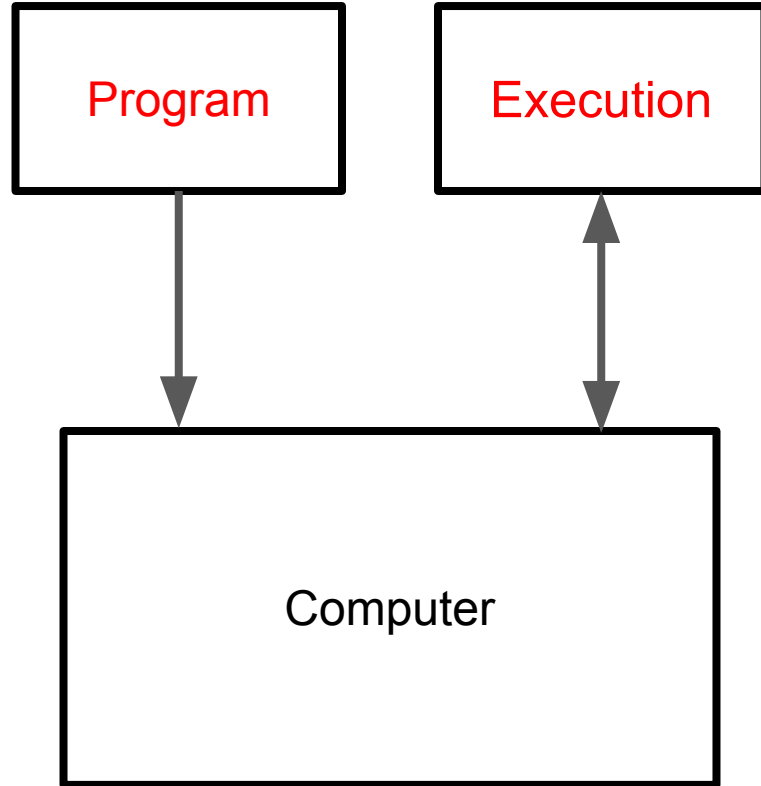
- What is a computer, actually.
- The widely spread memory model.
- The nerdier memory model*.
- Ok let's hack* Google!!1!

Current memory model



- Behaviour:
 - Constant latency, regardless of size
 - Constant bandwidth, regardless of size

The Arduino has Flash and SRAM*



- Theoretically, it follows our model
- Are flash and RAM equal?
- Are latency/bandwidth really constant?

General procedure to test memory

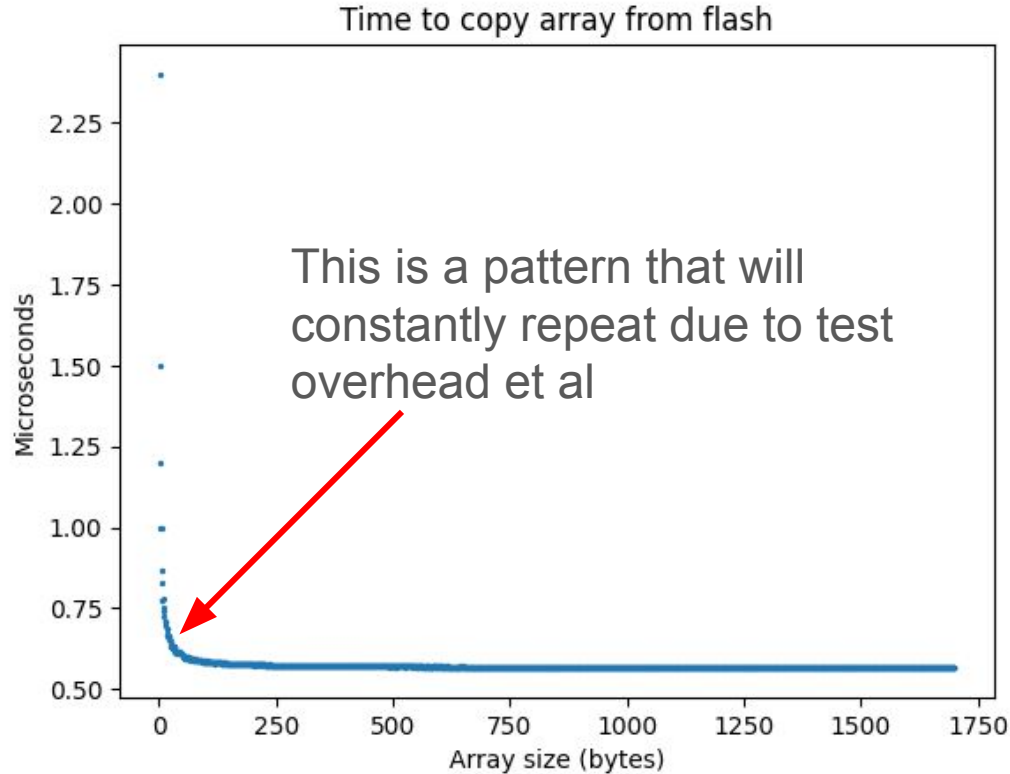
- Repeat the test several times to get rid of cold effects
- Test for increasingly large regions of memory
- Normalize the result to a per element basis

Copying flash to RAM: the code following the procedures

```
const int MAXN = 1842;
const byte flash[MAXN];

void setup() {
    // put your setup code here, to run once:
    byte arr[MAXN];
    Serial.begin(2000000);
    srand(analogRead(A0));
    for (int i = 1; i < MAXN; ++i) {
        long long ini = micros();
        for (int j = 0; j < 20; ++j) {
            memcpy_P(arr, &(flash), (sizeof arr[0])*(long long)i);
        }
        long long end = micros();
        Serial.print(i);
        Serial.print(", ");
        Serial.println((double)(end - ini));
    }
}
```

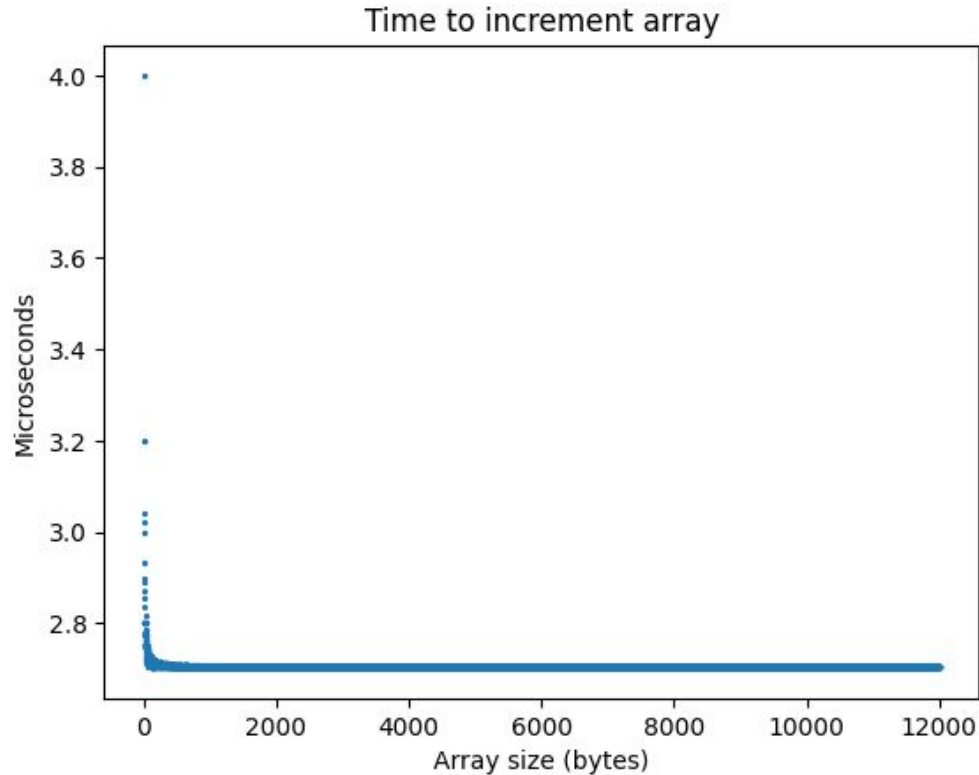
Copying flash to SRAM: the results are (not!) exciting



Flash bandwidth code: copy from flash each element

```
void setup() {  
  // put your setup code here, to run once:  
  Serial.begin(2000000);  
  srand(analogRead(A0));  
  for (int i = 1; i < MAXN; i+=1) {  
    byte chcksum = 1;  
    long long ini = micros();  
    for (int j = 0; j < 5; ++j) {  
      for (int k = 0; k < i; ++k) {  
        byte add; memcpy_P(&add, &arrays[k], sizeof add);  
        chcksum ^= add;  
      }  
    }  
    long long end = micros();  
    Serial.println(chcksum);  
    //Serial.println(freeMemory());  
    Serial.print(i);  
    Serial.print(", ");  
    Serial.println((double)(end - ini));  
  }  
}
```

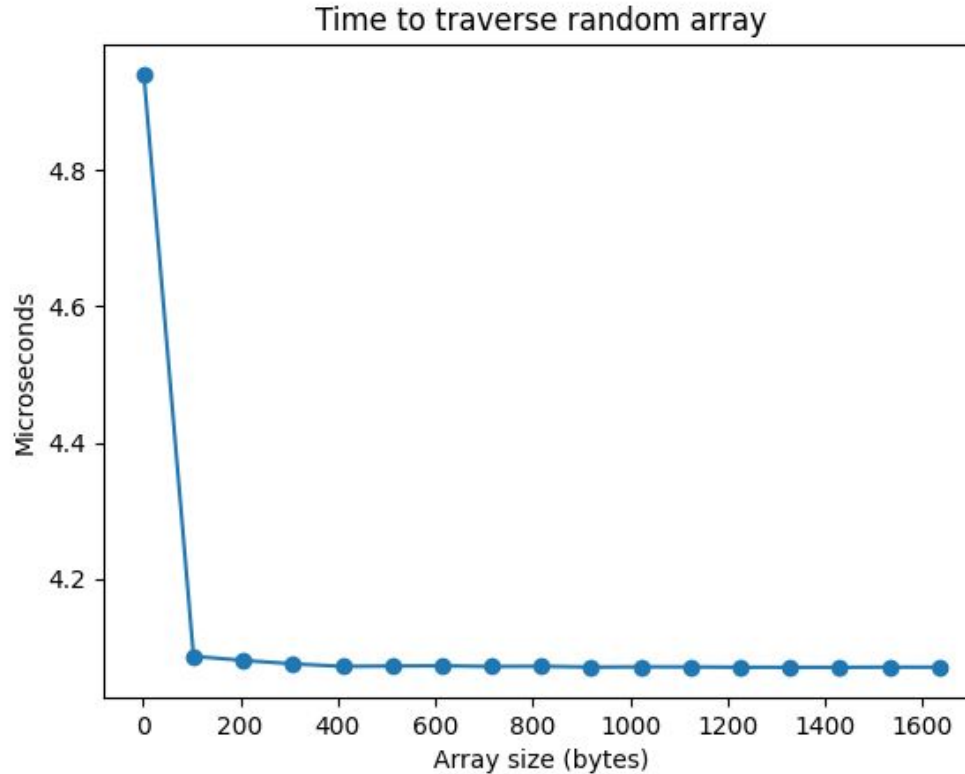
Flash bandwidth is also (not!!) exciting



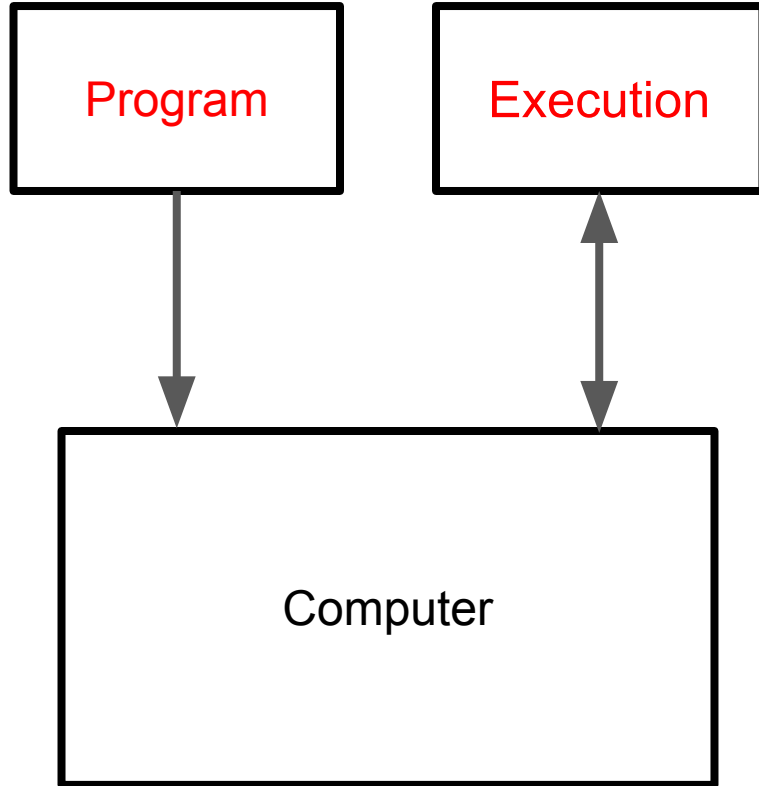
Flash latency code: iterate randomly through array

```
Serial.begin(2000000);
for (int i = 0; i < 17; ++i) {
  //memcpy_P(arr, (arrays[i]), sizeof arr);
  int count = 0;
  long long ini = micros();
  for (int j = 0; j < 20; ++j) {
    int k = 0;
    int add; memcpy_P(&add, &arrays[i][k], sizeof add);
    for (; add != 0; memcpy_P(&add, &arrays[i][k], sizeof add)) {
      k = add;
      ++count;
    }
  }
  count /= 20;
  ++count;
  long long end = micros();
  Serial.print(count);
  Serial.print(", ");
  Serial.println((double)(end - ini));
}
```

Flash latency is not exciting (Arduinos really are simple)



The ESP32 has Flash and SRAM*

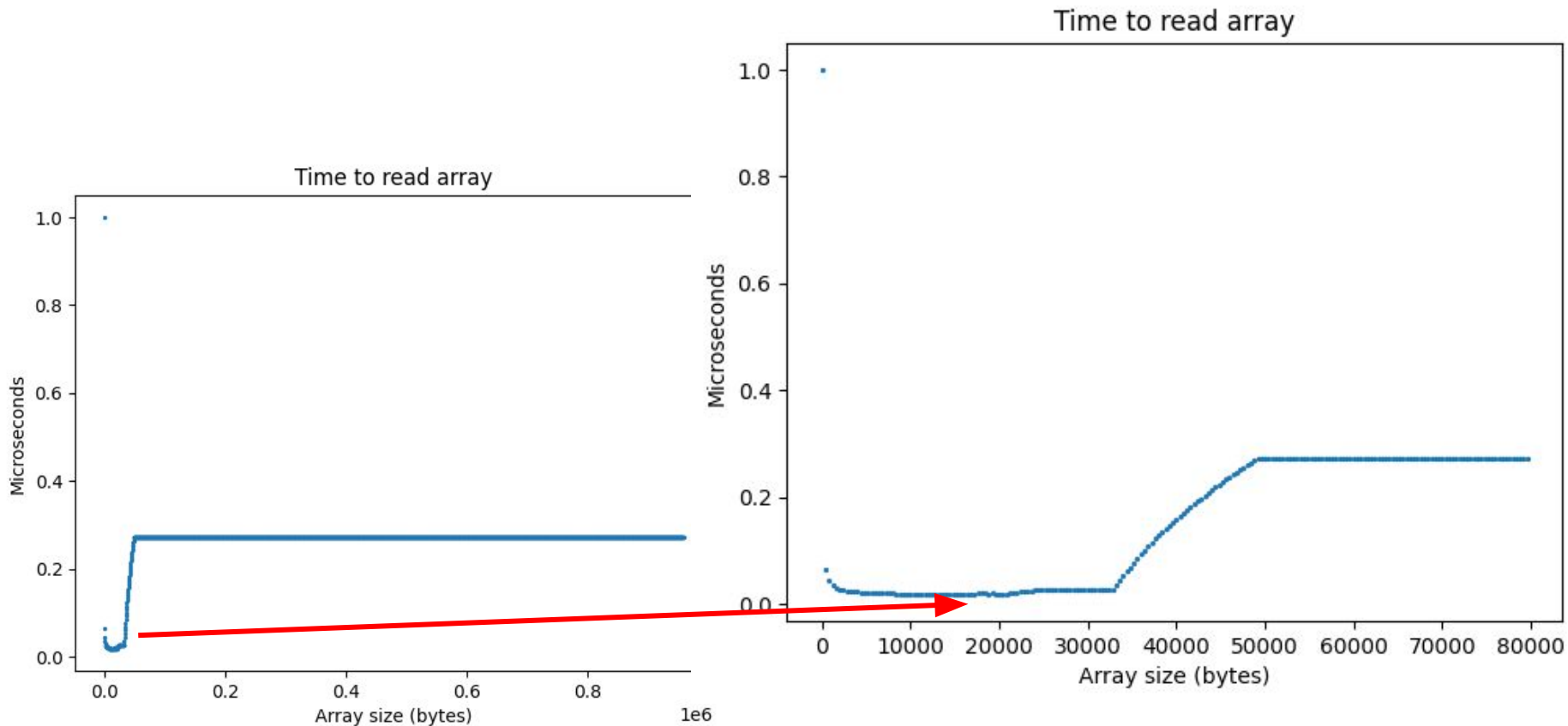


- Theoretically, it follows our model*
- Are latency/bandwidth really constant?

Flash bandwidth code

```
setCpuFrequencyMhz(240);
Serial.begin(1000000);
srand(analogRead(A0));
for (int i = 1; i < MAXN; i+=100) {
    int chcksum = 1;
    long long ini = micros();
    for (int j = 0; j < 5; ++j) {
        for (int k = 0; k < i; ++k) {
            chcksum ^= arrays[k];
        }
    }
    long long end = micros();
    Serial.println(chcksum);
    Serial.print(i);
    Serial.print(", ");
    Serial.println((double)(end - ini));
}
```

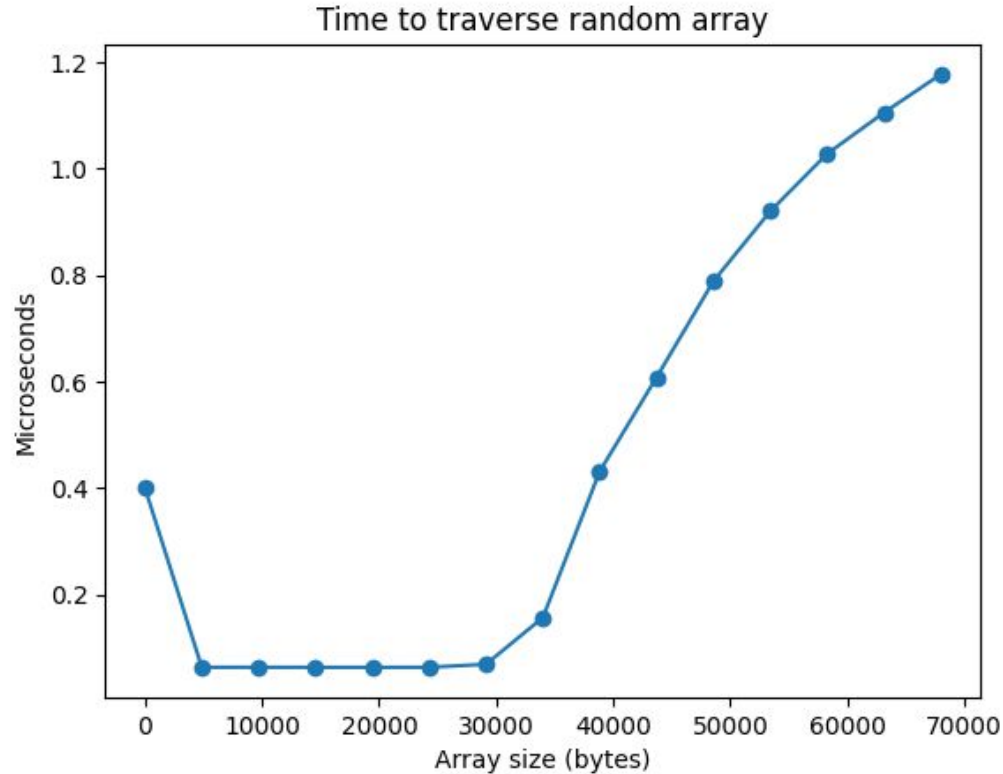
Bandwidth exhibits a weird dip



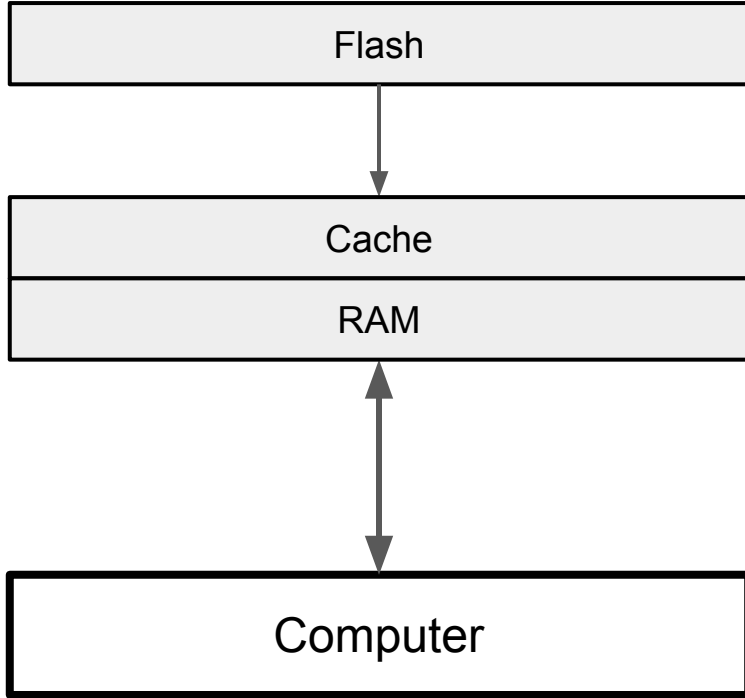
Flash latency code

```
setCpuFrequencyMhz(240);
srand(analogRead(A0));
Serial.begin(1000000);
for (int i = 0; i < 20; ++i) {
    int count = 0;
    long long ini = micros();
    for (int j = 0; j < 20; ++j) {
        for (int k = 0; arrays[i][k] != 0; k = arrays[i][k]) {
            ++count;
        }
    }
    count /= 20;
    ++count;
    long long end = micros();
    Serial.print(count);
    Serial.print(", ");
    Serial.println((double)(end - ini));
}
```

Latency appears to increase after the same size



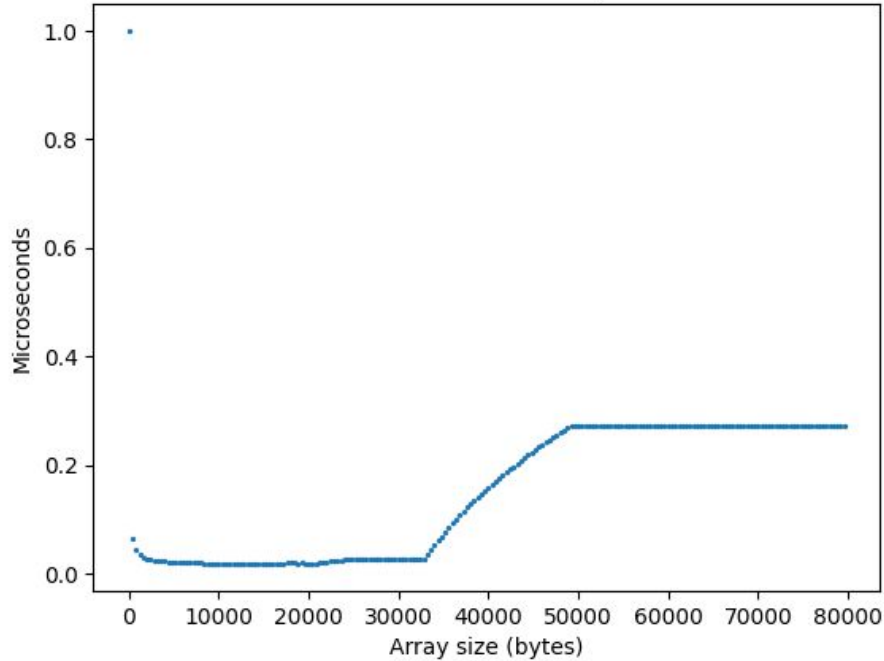
Cache is a faster, much smaller memory



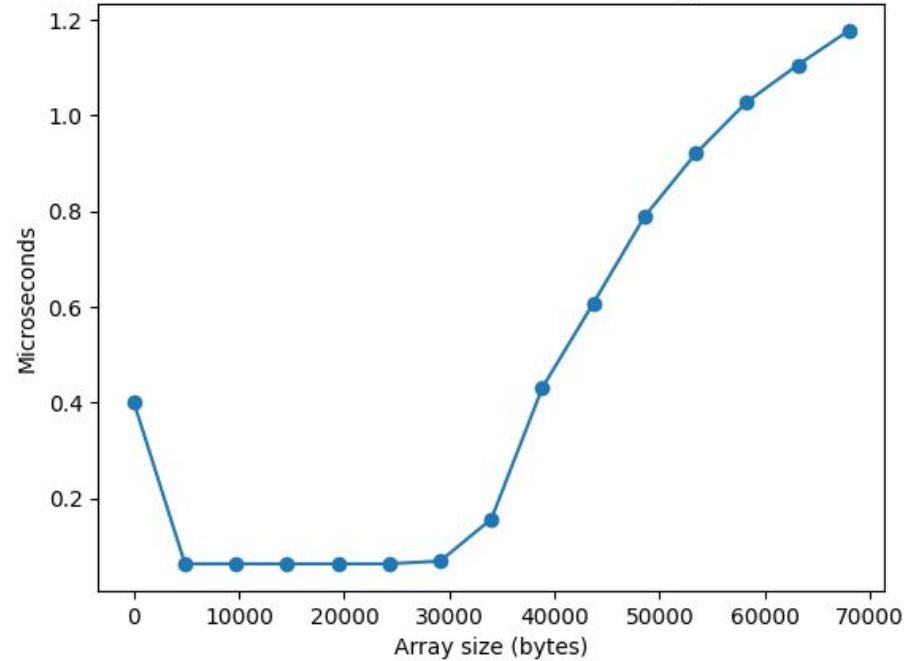
- Every time the computer retrieves something from memory, it is stored in cache
- Cache is small, so if it doesn't have space old data is kicked out
- Cache has much higher bandwidth and much lower latency

Cache explains the ESP32 behaviour (32kB)

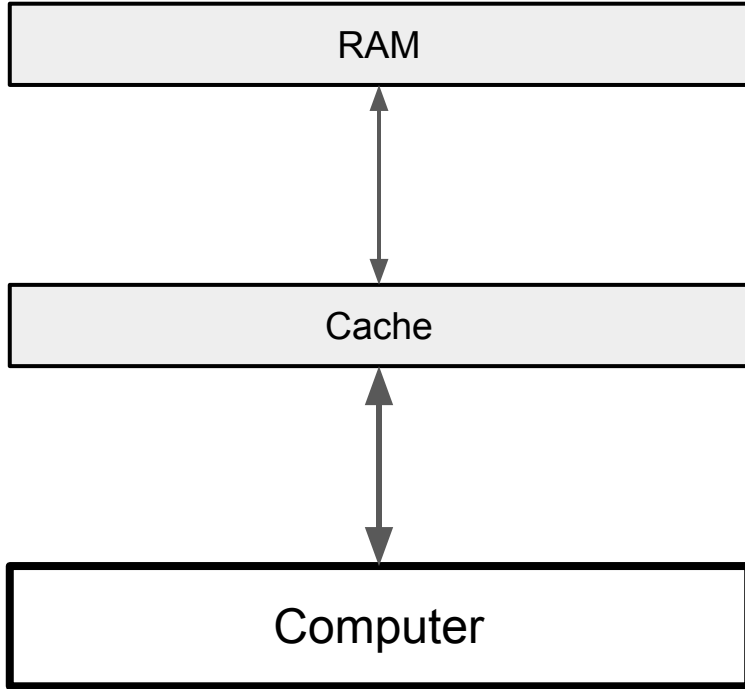
Time to read array



Time to traverse random array



The Raspberry has cache and RAM

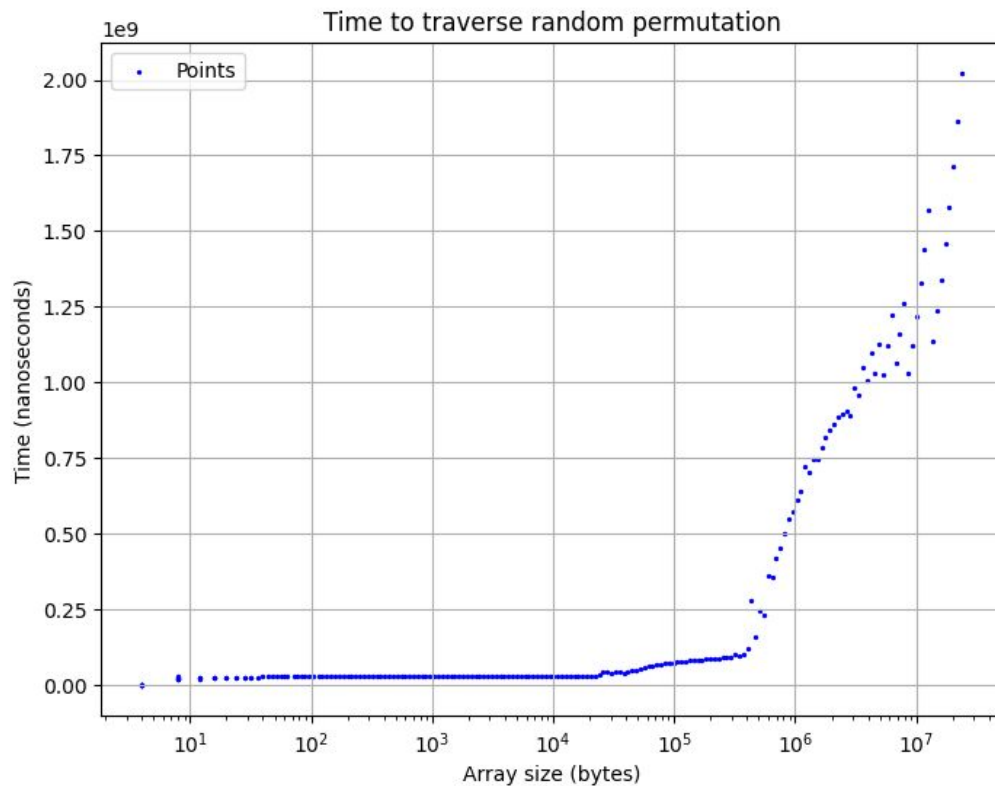


- Theoretically, it follows our model*
- Do we see the steps like the in the ESP32 graphs?

Memory latency code

```
volatile int count = 0;
auto ini = chrono::high_resolution_clock::now();
for(int i = 0; i < N/len; ++i){
    for(int j = 0; arr[j] != 0; j = arr[j]){
        ++count;
    }
}
auto end = chrono::high_resolution_clock::now();
long int elapsed_ns = chrono::duration_cast<chrono::nanoseconds>(end - ini).count();
if(file.is_open()){
    file << (int) len << ", " << elapsed_ns << endl;
}
```

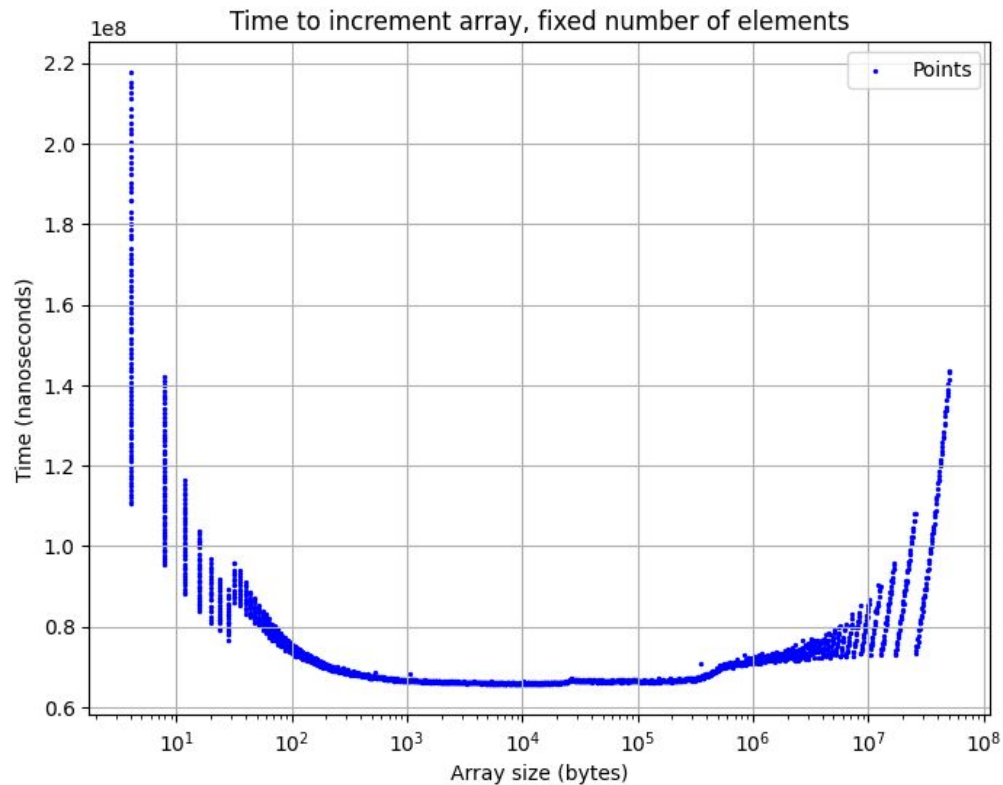

Memory latency results seem consistent



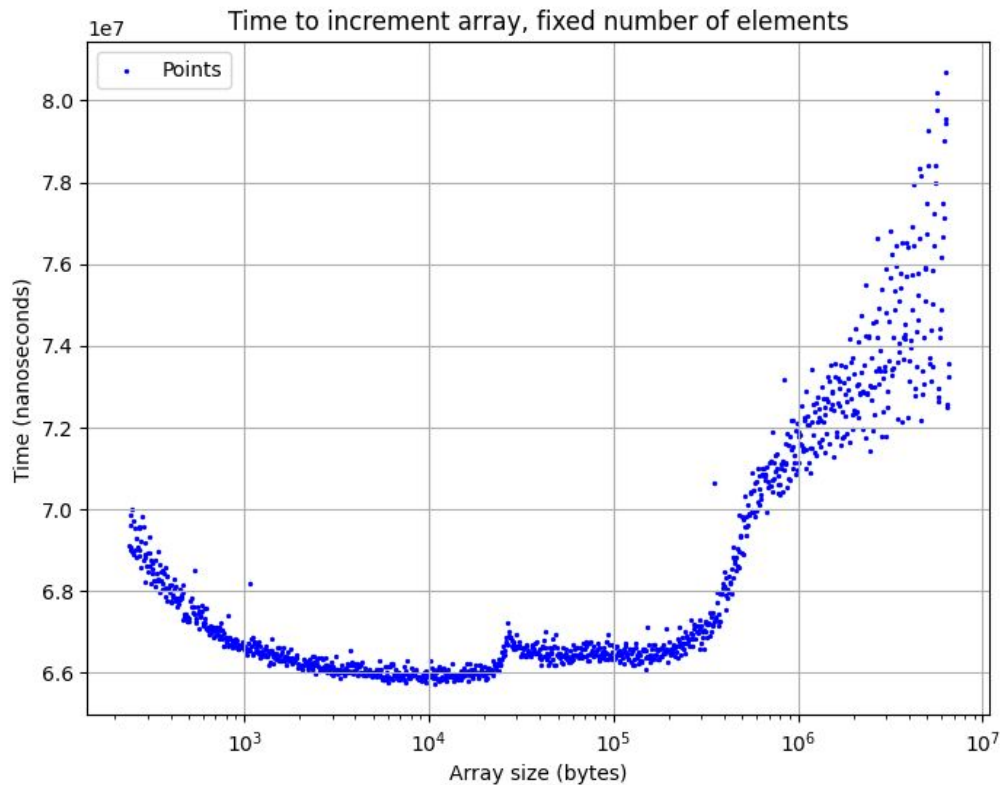
Memory bandwidth code

```
auto ini = chrono::high_resolution_clock::now();
__sync_synchronize();
for(int i = 0; i < N/len; ++i){
    for(int j = 0; j < len; ++j){
        ++arr[j];
    }
}
auto end = chrono::high_resolution_clock::now();
long int elapsed_ns = chrono::duration_cast<chrono::nanoseconds>(end - ini).count();
if(file.is_open()){
    file << (int) len << ", " << elapsed_ns << endl;
}
```

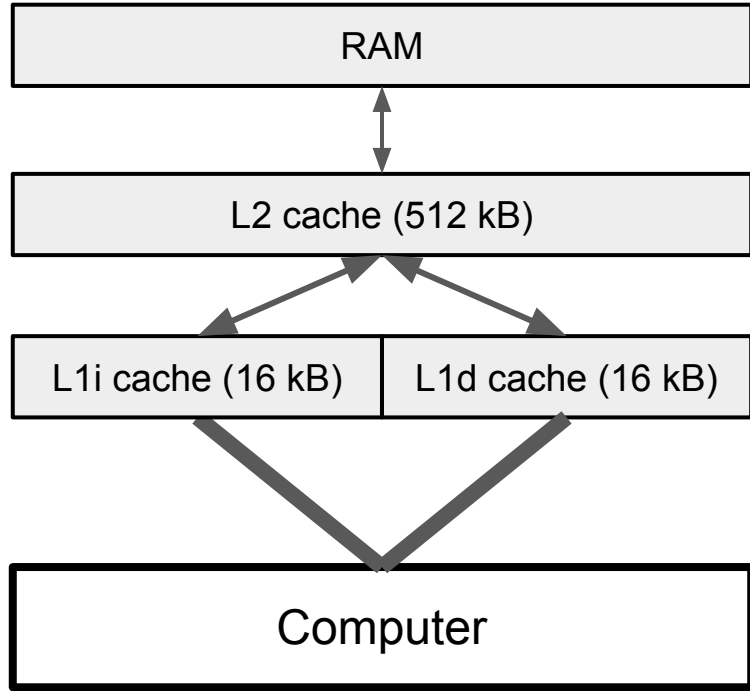
Memory bandwidth seems normal-ish



Or not! The zoomed in version shows more steps

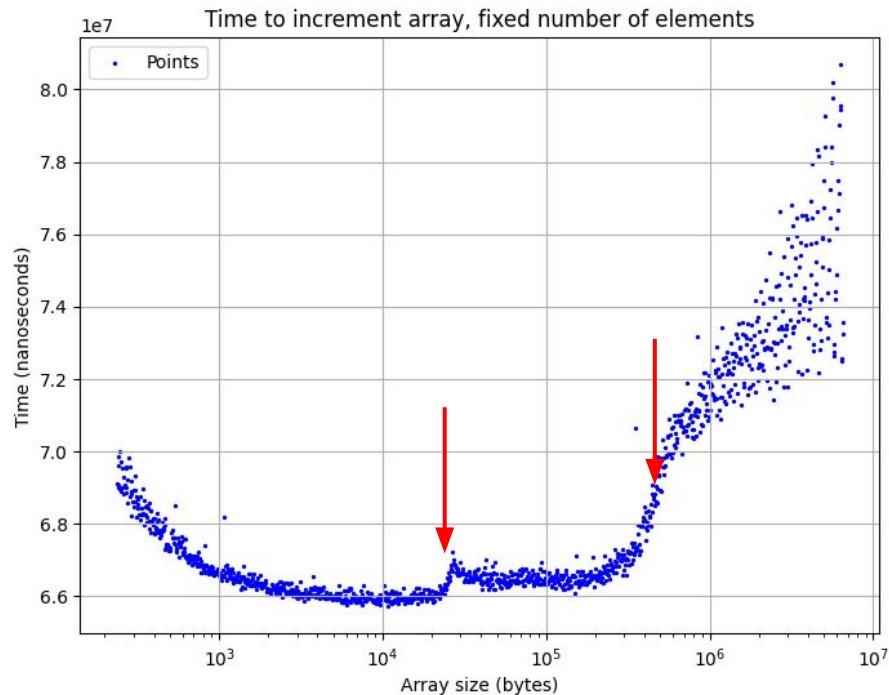
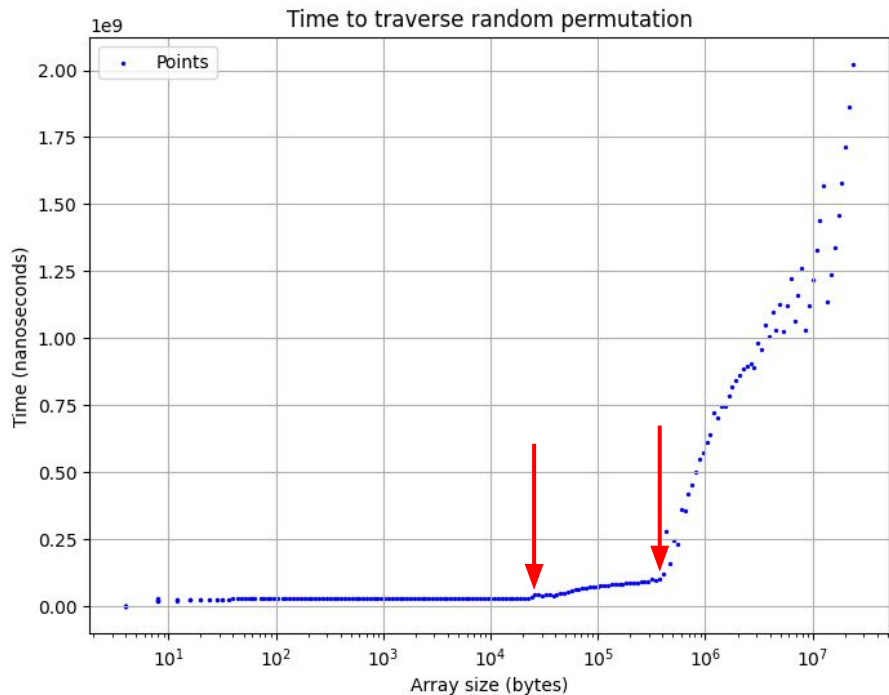


The Raspberry has two levels of cache!

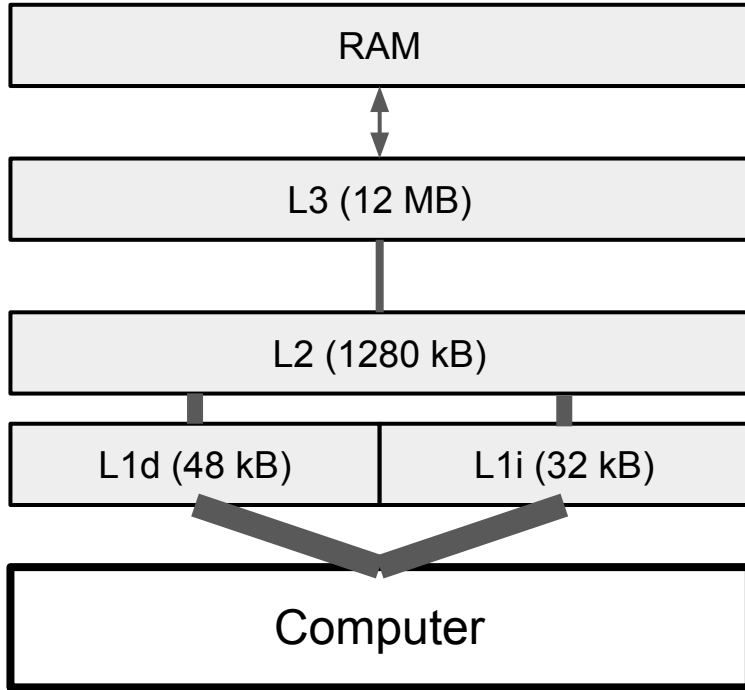


- Data first arrives in L1
- Old data is expelled to L2, only then to RAM
- If something from L2 is called, it returns to L1

This agrees with our data closely (two steps)



x86 computers have RAM and multiple layers of cache

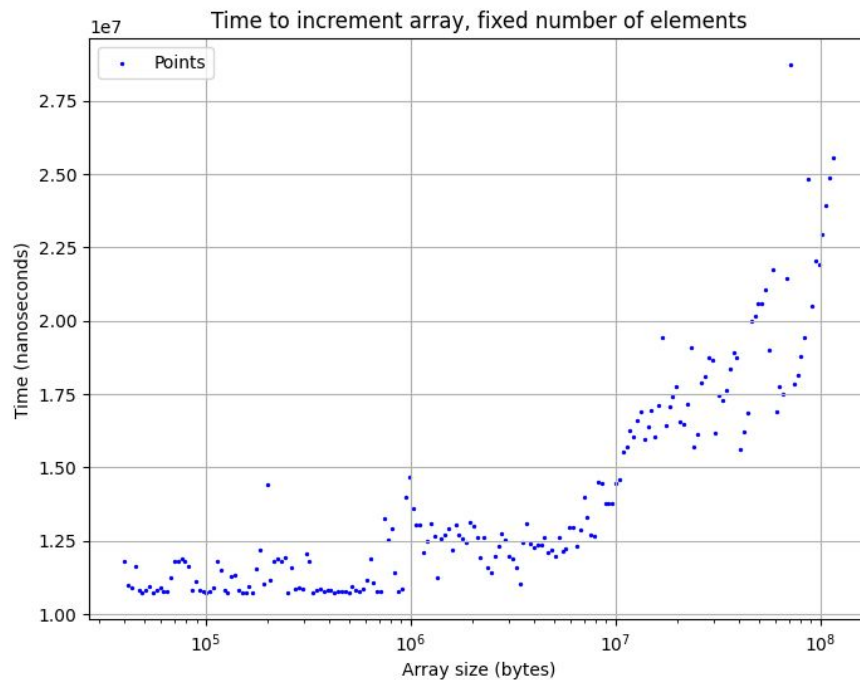
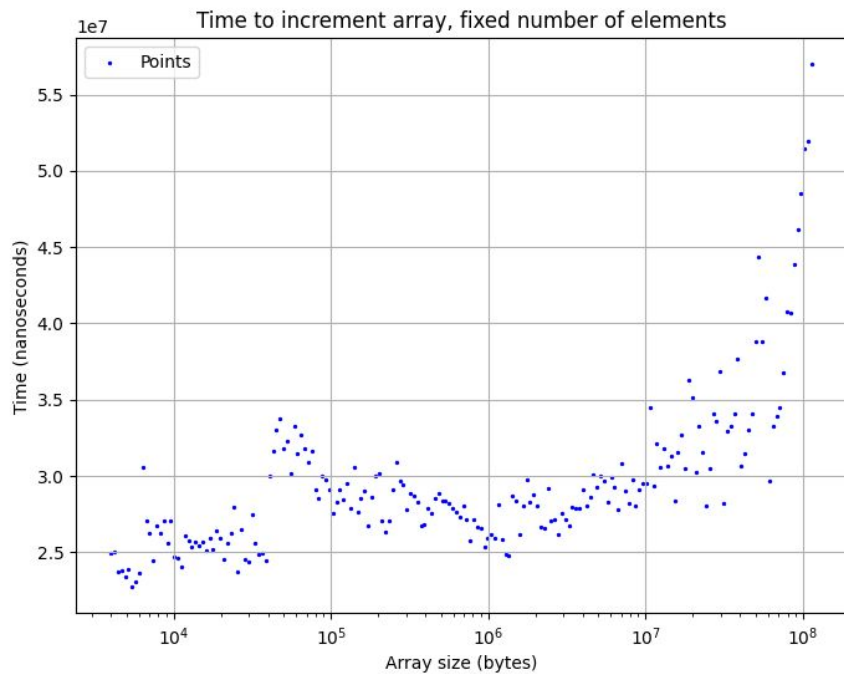


- Theoretically, it follows our model*
- Do we see the steps like the in the Raspberry graphs?
- Does accessing data close to each other yield the same bandwidth?

Bandwidth code

```
for(double len = 1e4; len < total; len *= pow(10, (log10(total) - 4)/200)){
    auto ini = chrono::high_resolution_clock::now();
    __sync_synchronize();
    for(int i = 0; i < N/len; ++i){
        for(int j = 0; j < len; ++j){
            ++arr[j];
        }
    }
    auto end = chrono::high_resolution_clock::now();
    long int elapsed_ns = chrono::duration_cast<chrono::nanoseconds>(end - ini).count();
    if(file.is_open()){
        file << (int) len << ", " << elapsed_ns << endl;
    }
}
```

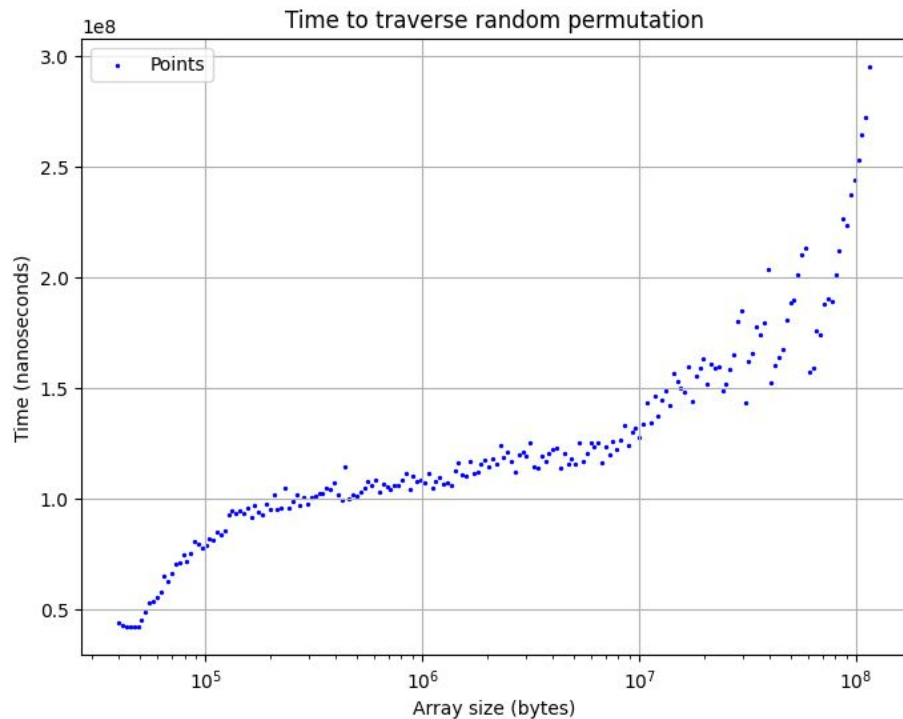
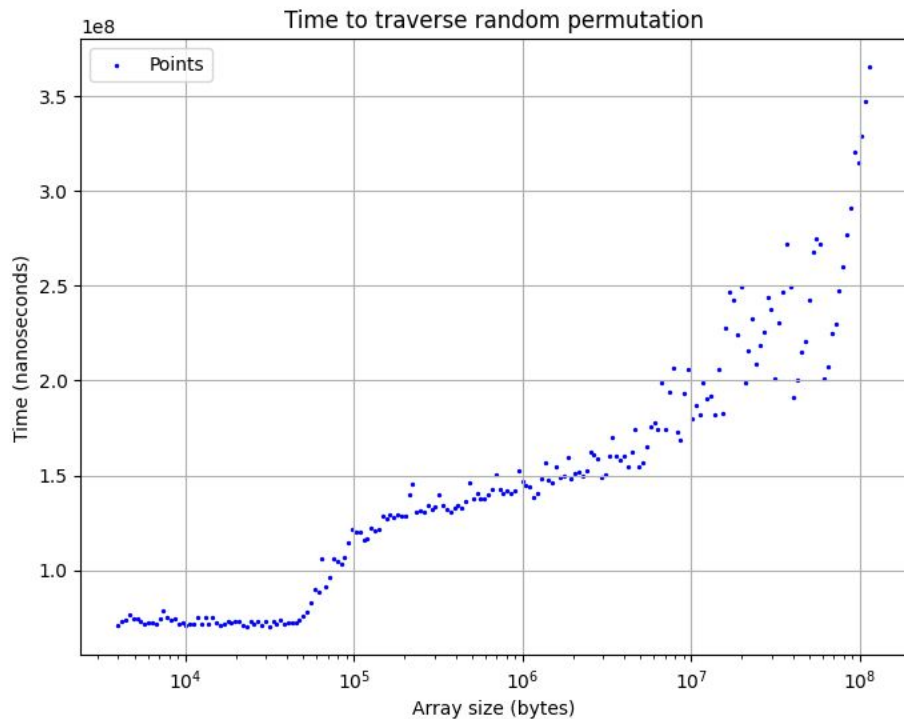

Bandwidth results



Latency code

```
volatile int count = 0;
auto ini = chrono::high_resolution_clock::now();
for(int i = 0; i < N/len; ++i){
    for(int j = 0; arr[j] != 0; j = arr[j]){
        ++count;
    }
}
auto end = chrono::high_resolution_clock::now();
long int elapsed_ns = chrono::duration_cast<chrono::nanoseconds>(end - ini).count();
if(file.is_open()){
    file << (int) len << ", " << elapsed_ns << endl;
}
```

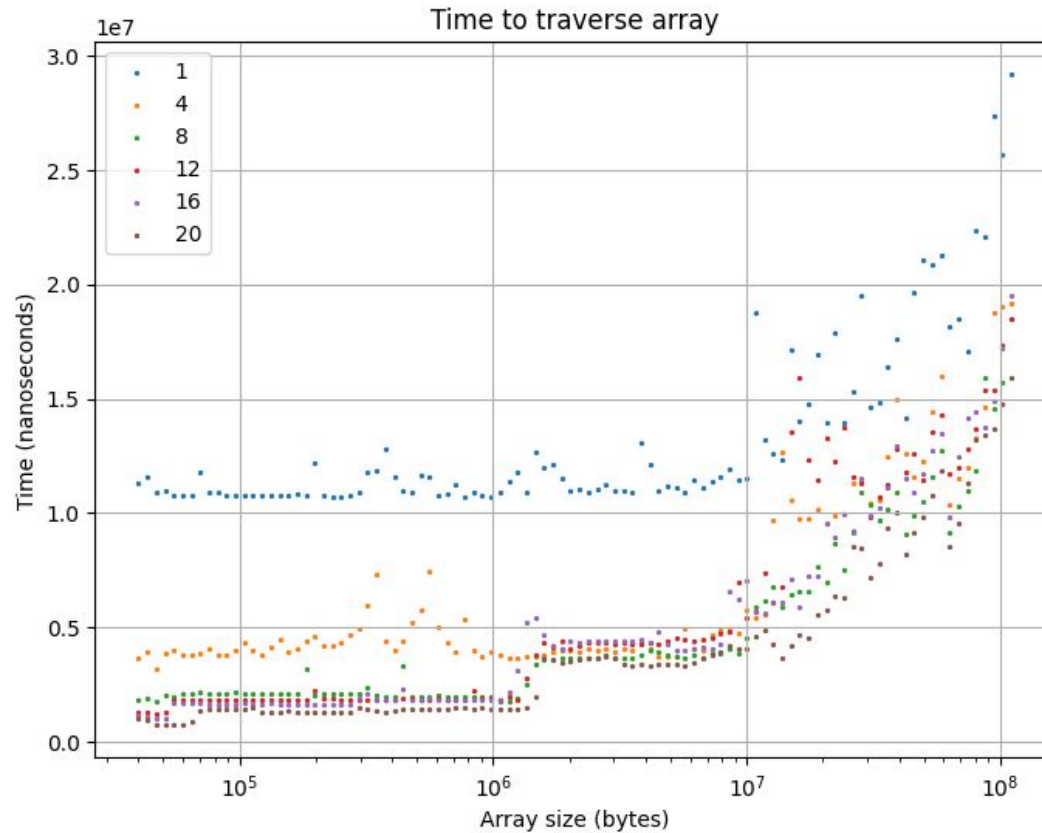
Latency results



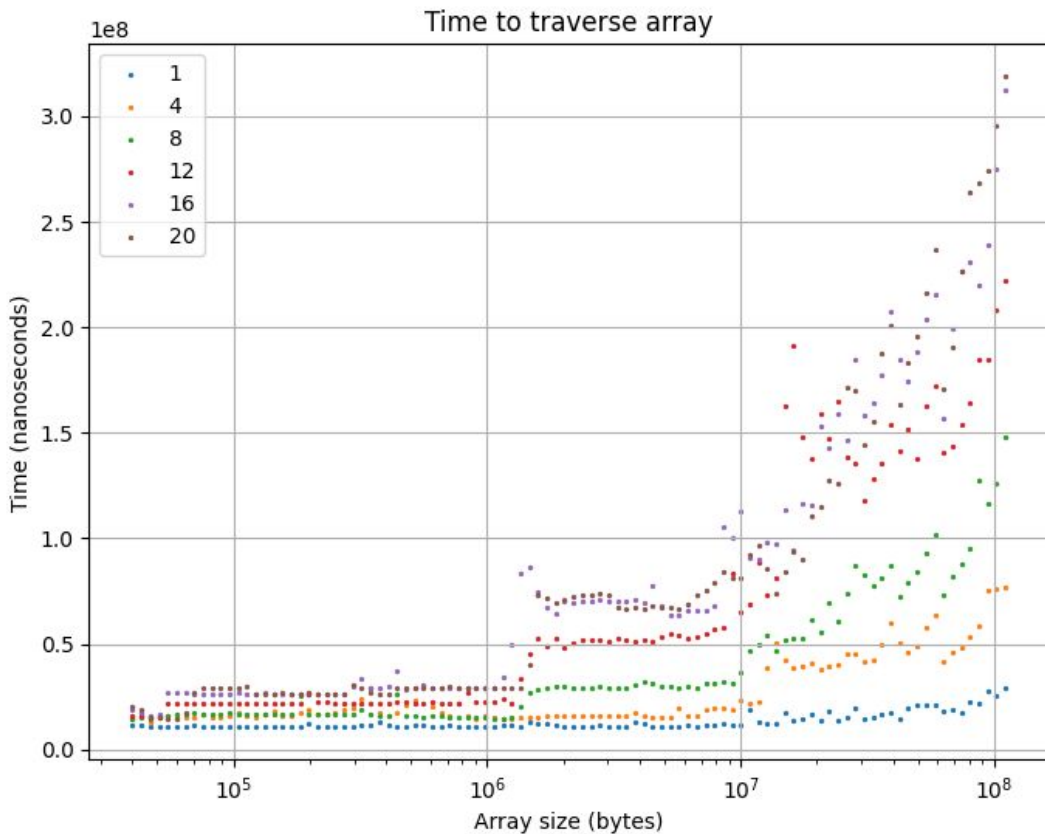
Strides code

```
for(int stride = 1; stride <= 20; ++stride){
    for(double len = 1e4; len < total; len *= pow(10, (log10(total) - 4)/100)){
        auto ini = chrono::high_resolution_clock::now();
        __sync_synchronize();
        for(int i = 0; i < N/len; ++i){
            for(int j = 0; j < len; j += stride){
                ++arr[j];
            }
        }
        auto end = chrono::high_resolution_clock::now();
        long int elapsed_ns = chrono::duration_cast<chrono::nanoseconds>(end - ini).count();
        if(file.is_open()){
            file << stride << ", " <<(int) len << ", " << elapsed_ns << endl;
        }
    }
}
```

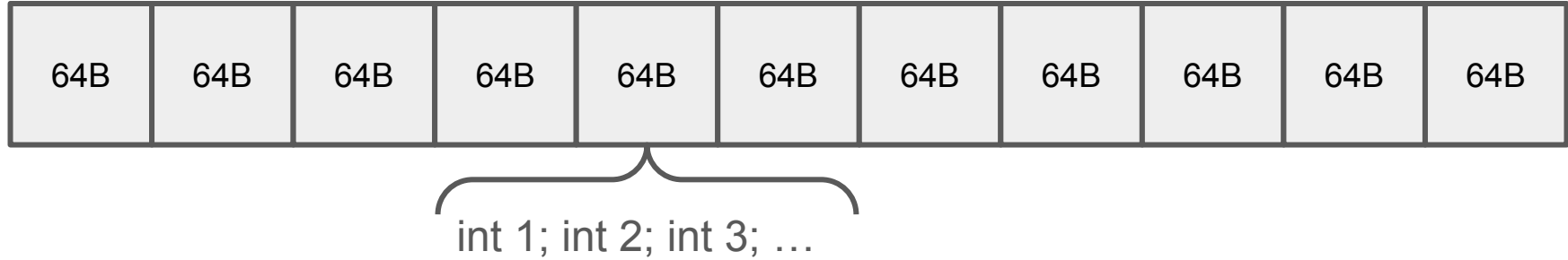
Strides bandwidth



Strides bandwidth, normalized per element



Cache is fetched in batches of 64B



- Larger strides use more cache lines
- The computer processes integers faster than they can be fetched
- The computer ends up bottlenecked by the fetching stage

Presentation outline

- What is a computer, actually.
- The widely spread memory model.
- The nerdier memory model*.
- Ok let's hack* Google!!1!

Computers hide latency with speculative execution

Needs to retrieve
from cache, possibly
even RAM

```
if(arr[5] > 10){  
    //blabla  
}
```

The computer predicts
the outcome so it
doesn't have to wait

When the value arrives, the computer undoes its
execution if the prediction was wrong

The spectre hardware vulnerability is very serious

- Discovered in 2018 by Google researchers and others
- Affects ~every Intel CPU since 1995-ish, not yet solved but mitigated
- Programs exploit the memory system and speculative execution to learn what is in memory they don't have access to (passwords, etc.)

Speculative execution can promote things to L1


```
if(arr[5] > 10){  
    int aaa = arr[75937509];  
}
```



Even if predicted wrong,
arr[75937509] will remain in cache

Spectre checks access latency

```
if(arr[5] > 10){  
    int oops = mwaHaHaHa[arr[75937509]];  
}
```



Even if the computer realises `arr[75937509]` shouldn't be accessed, `mwaHaHaHa[arr[75937509]]` will have been promoted. But accessing `mwaHaHaHa` later is fine!

Let's hack google!!!1!