Programming in C

Lecture 6: The Memory Hierarchy and Cache Optimization

Dr Neel Krishnaswami

Michaelmas Term 2017-2018

Three Simple C Functions

```
void increment_every(int *}array)
  for (int i = 0; i < BIG_NUMBER; i += 1) {
    array[i] = 0;
}
void increment_8th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 8)
    array[i] = 0;
}
void increment_16th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 16)
    array[i] = 0;
}
```

Three Simple C Functions

```
void increment_every(int *}array)
  for (int i = 0; i < BIG_NUMBER; i += 1) {
    array[i] = 0;
}
void increment_8th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 8)
    array[i] = 0;
}
void increment_16th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 16)
    array[i] = 0;
}
```

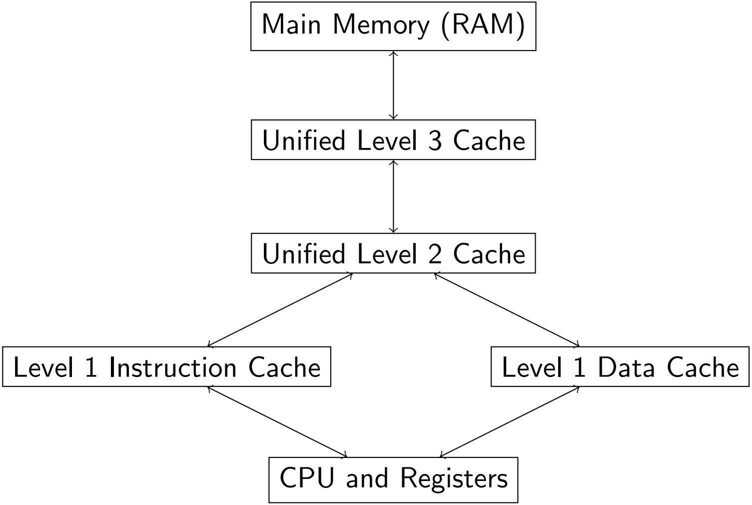
- Which runs faster?

Three Simple C Functions

```
void increment_every(int *}array)
  for (int i = 0; i < BIG_NUMBER; i += 1) {
    array[i] = 0;
}
void increment_8th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 8)
    array[i] = 0;
}
void increment_16th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 16)
    array[i] = 0;
}
```

- Which runs faster?
- . . . and by how much?

The Memory Hierarchy



Latencies in the Memory Hierarchy

Access Type	Cycles	Time	Human Scale
L1 cache reference	≈4	1.3 ns	1s
L2 cache reference	≈10	4 ns	3s
L3 cache reference, unshared	≈40	13 ns	10s
L3 cache reference, shared	≈65	20 ns	16s
Main memory reference	≈300	100 ns	80s

Latencies in the Memory Hierarchy

Access Type	Cycles	Time	Human Scale
L1 cache reference	≈4	1.3 ns	1s
L2 cache reference	≈10	4 ns	3s
L3 cache reference, unshared	≈40	13 ns	10s
L3 cache reference, shared	≈65	20 ns	16s
Main memory reference	≈300	100 ns	80s

- Accesses to main memory are *slow*

Latencies in the Memory Hierarchy

Access Type	Cycles	Time	Human Scale
L1 cache reference	≈4	1.3 ns	1s
L2 cache reference	≈10	4 ns	3s
L3 cache reference, unshared	≈40	13 ns	10s
L3 cache reference, shared	≈65	20 ns	16s
Main memory reference	≈300	100 ns	80s

- Accesses to main memory are *slow*
- This can dominate performance!

How Caches Work

When a CPU looks up an address. . . :

How Caches Work

When a CPU looks up an address. . . :

1. It looks up the address in the cache

How Caches Work

When a CPU looks up an address. . . :

1. It looks up the address in the cache
2. If present, this is a *cache hit* (cheap!)

How Caches Work

When a CPU looks up an address. . . :

1. It looks up the address in the cache
2. If present, this is a *cache hit* (cheap!)
3. If absent, this is a *cache miss*

How Caches Work

When a CPU looks up an address...:

1. It looks up the address in the cache
2. If present, this is a *cache hit* (cheap!)
3. If absent, this is a *cache miss*
   3.1 The address is then looked up in main memory (expensive!)

How Caches Work

When a CPU looks up an address. . . :

1. It looks up the address in the cache
2. If present, this is a *cache hit* (cheap!)
3. If absent, this is a *cache miss*
    3.1 The address is then looked up in main memory (expensive!)
    3.2 The address/value pair is then stored in the cache

How Caches Work

When a CPU looks up an address. . . :

1. It looks up the address in the cache
2. If present, this is a *cache hit* (cheap!)
3. If absent, this is a *cache miss*
    3.1 The address is then looked up in main memory (expensive!)
    3.2 The address/value pair is then stored in the cache
    3.3 . . . along with the next 64 bytes (typically) of memory

How Caches Work

When a CPU looks up an address. . . :

1. It looks up the address in the cache
2. If present, this is a *cache hit* (cheap!)
3. If absent, this is a *cache miss*
   3.1 The address is then looked up in main memory (expensive!)
   3.2 The address/value pair is then stored in the cache
   3.3 . . . along with the next 64 bytes (typically) of memory
   3.4 This is a *cache line* or *cache block*

Locality: Taking advantage of caching

Caching is most favorable:

Locality: Taking advantage of caching

Caching is most favorable:

- Each piece of data the program works on is near (in RAM) the address of the last piece of data the program worked on.

Locality: Taking advantage of caching

Caching is most favorable:

- Each piece of data the program works on is near (in RAM) the address of the last piece of data the program worked on.
- This is the *principle of locality*

Locality: Taking advantage of caching

Caching is most favorable:

- Each piece of data the program works on is near (in RAM) the address of the last piece of data the program worked on.
- This is the *principle of locality*
- Performance engineering involves redesigning data structures to take advantage of locality.

Pointers Are Expensive

Consider the following Java linked list implementation

```
class List<T> {
  public T head;
  public List<T> tail;

  public List(T head, List<T> tail) {
    this.head = head;
    this.tail = tail;
  }
}
```

Pointers Are Expensive

It corresponds to the following C code:

```
typedef struct List* list_t;
struct List {
  void *head;
  list_t tail;
};
list_t list_cons(void *head, list_t tail) {
  list_t result = malloc(sizeof(struct list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

Pointers Are Expensive

It corresponds to the following C code:

```
typedef struct List* list_t;
struct List {
  void *head;
  list_t tail;
};
list_t list_cons(void *head, list_t tail) {
  list_t result = malloc(sizeof(struct list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

- We use `void *` for genericity, but this introduces pointer indirections.

Pointers Are Expensive

It corresponds to the following C code:

```
typedef struct List* list_t;
struct List {
  void *head;
  list_t tail;
};
list_t list_cons(void *head, list_t tail) {
  list_t result = malloc(sizeof(struct list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

- We use `void *` for genericity, but this introduces pointer indirections.
- This can get expensive!

Specializing the Representation

Suppose we use a list at a `Data *` type:

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

struct List {
  Data *head;
  struct List *tail;
};
```

Technique #1: Intrusive Lists

We can try changing the list representation to:

```
typedef struct intrusive_list ilist_t;
struct intrusive_list {
  Data head;
  ilist_t tail;
};
ilist_t ilist_cons(Data head, ilist_t tail) {
  list_t result = malloc(sizeof(struct intrusive_list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

Technique #1: Intrusive Lists

We can try changing the list representation to:

```
typedef struct intrusive_list ilist_t;
struct intrusive_list {
  Data head;
  ilist_t tail;
};
ilist_t ilist_cons(Data head, ilist_t tail) {
  list_t result = malloc(sizeof(struct intrusive_list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

- The indirection in the head is removed

Technique #1: Intrusive Lists

We can try changing the list representation to:

```
typedef struct intrusive_list ilist_t;
struct intrusive_list {
  Data head;
  ilist_t tail;
};
ilist_t ilist_cons(Data head, ilist_t tail) {
  list_t result = malloc(sizeof(struct intrusive_list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

- The indirection in the head is removed
- But we had to use a specialized representation

Technique #1: Intrusive Lists

We can try changing the list representation to:

```
typedef struct intrusive_list ilist_t;
struct intrusive_list {
  Data head;
  ilist_t tail;
};
ilist_t ilist_cons(Data head, ilist_t tail) {
  list_t result = malloc(sizeof(struct intrusive_list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

- The indirection in the head is removed
- But we had to use a specialized representation
- Can no longer use generic linked list routines

Technique #2: Lists of Structs to Arrays of Structs

Linked lists are expensive:

Technique #2: Lists of Structs to Arrays of Structs

Linked lists are expensive:

1. Following a tail pointer can lead to *cache miss*

Technique #2: Lists of Structs to Arrays of Structs

Linked lists are expensive:

1. Following a tail pointer can lead to *cache miss*
2. Cons cells requiring storing a tail pointer...

Technique #2: Lists of Structs to Arrays of Structs

Linked lists are expensive:

1. Following a tail pointer can lead to *cache miss*
2. Cons cells requiring storing a tail pointer...
3. This reduces the number of data elements that fit in a cache line

Technique #2: Lists of Structs to Arrays of Structs

Linked lists are expensive:

1. Following a tail pointer can lead to *cache miss*
2. Cons cells requiring storing a tail pointer...
3. This reduces the number of data elements that fit in a cache line
4. This decreases data density, and increases *cache miss rate*

Technique #2: Lists of Structs to Arrays of Structs

Linked lists are expensive:

1. Following a tail pointer can lead to *cache miss*
2. Cons cells requiring storing a tail pointer. . .
3. This reduces the number of data elements that fit in a cache line
4. This decreases data density, and increases *cache miss rate*
5. Replace `ilist_t` with `Data[]`!

Technique #2: Lists of Structs to Arrays of Structs

We can try changing the list representation to:

```
Data *iota_array(int n) {
  Data *a = malloc(n * sizeof(Data));
  for (int i = 0; i < n; i++) {
    a[i].i = i;
    a[i].d =  1.0;
    a[i].c = 'x';
  }
  return a;
}
```

Technique #2: Lists of Structs to Arrays of Structs

We can try changing the list representation to:

```
Data *iota_array(int n) {
  Data *a = malloc(n * sizeof(Data));
  for (int i = 0; i < n; i++) {
    a[i].i = i;
    a[i].d =  1.0;
    a[i].c = 'x';
  }
  return a;
}
```

- No longer store tail pointers

Technique #2: Lists of Structs to Arrays of Structs

We can try changing the list representation to:

```
Data *iota_array(int n) {
  Data *a = malloc(n * sizeof(Data));
  for (int i = 0; i < n; i++) {
    a[i].i = i;
    a[i].d =  1.0;
    a[i].c = 'x';
  }
  return a;
}
```

- No longer store tail pointers
- Every element comes after previous element in memory

Technique #2: Lists of Structs to Arrays of Structs

We can try changing the list representation to:

```
Data *iota_array(int n) {
  Data *a = malloc(n * sizeof(Data));
  for (int i = 0; i < n; i++) {
    a[i].i = i;
    a[i].d =  1.0;
    a[i].c = 'x';
  }
  return a;
}
```

- No longer store tail pointers
- Every element comes after previous element in memory
- Can no longer incrementally build lists

Technique #2: Lists of Structs to Arrays of Structs

We can try changing the list representation to:

```
Data *iota_array(int n) {
  Data *a = malloc(n * sizeof(Data));
  for (int i = 0; i < n; i++) {
    a[i].i = i;
    a[i].d =  1.0;
    a[i].c = 'x';
  }
  return a;
}
```

- No longer store tail pointers
- Every element comes after previous element in memory
- Can no longer incrementally build lists
- Have to know size up-front

Technique #3: Arrays of Structs to Struct of Arrays

Suppose we have an operation

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

void traverse_array(int n, Data *a) {
  for (int i = 0; i < n; i++)
    a[i].c += 'y';
}
```

Technique #3: Arrays of Structs to Struct of Arrays

Suppose we have an operation

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

void traverse_array(int n, Data *a) {
  for (int i = 0; i < n; i++)
    a[i].c += 'y';
}
```

- Note that we are only modifying character field c.

Technique #3: Arrays of Structs to Struct of Arrays

Suppose we have an operation

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

void traverse_array(int n, Data *a) {
  for (int i = 0; i < n; i++)
    a[i].c += 'y';
}
```

- Note that we are only modifying character field c.
- We have “hop over” the integer and double fields.

Technique #3: Arrays of Structs to Struct of Arrays

Suppose we have an operation

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

void traverse_array(int n, Data *a) {
  for (int i = 0; i < n; i++)
    a[i].c += 'y';
}
```

- Note that we are only modifying character field c.
- We have "hop over" the integer and double fields.
- So characters are at least 12, and probably 16 bytes apart.

Technique #3: Arrays of Structs to Struct of Arrays

Suppose we have an operation

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

void traverse_array(int n, Data *a) {
  for (int i = 0; i < n; i++)
    a[i].c += 'y';
}
```

- Note that we are only modifying character field c.
- We have “hop over” the integer and double fields.
- So characters are at least 12, and probably 16 bytes apart.
- This means only 4 characters in each cache line...

Technique #3: Arrays of Structs to Struct of Arrays

Suppose we have an operation

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

void traverse_array(int n, Data *a) {
  for (int i = 0; i < n; i++)
    a[i].c += 'y';
}
```

- Note that we are only modifying character field c.
- We have “hop over” the integer and double fields.
- So characters are at least 12, and probably 16 bytes apart.
- This means only 4 characters in each cache line. . .
- Optimally, 64 characters fit in each cache line. . .

Technique #3: Arrays of Structs to Struct of Arrays

```
typedef struct datavec *DataVec;
struct datavec {
  int *is;
  double *ds;
  char *cs;
};
```

Technique #3: Arrays of Structs to Struct of Arrays

```
typedef struct datavec *DataVec;
struct datavec {
  int *is;
  double *ds;
  char *cs;
};
```

- Instead of storing an array of structures...

Technique #3: Arrays of Structs to Struct of Arrays

```
typedef struct datavec *DataVec;
struct datavec {
  int *is;
  double *ds;
  char *cs;
};
```

- Instead of storing an array of structures...
- We store a struct of arrays

Technique #3: Arrays of Structs to Struct of Arrays

```
typedef struct datavec *DataVec;
struct datavec {
  int *is;
  double *ds;
  char *cs;
};
```

- Instead of storing an array of structures. . .
- We store a struct of arrays
- Now traversing just the cs is easy

Technique #3: Traversing Struct of Arrays

```c
void traverse_datavec(int n, DataVec d) {
  char *a = d->cs;
  for (int i = 0; i < n; i++) {
    a[i] += 'y';
  }
}
```

Technique #3: Traversing Struct of Arrays

```
void traverse_datavec(int n, DataVec d) {
  char *a = d->cs;
  for (int i = 0; i < n; i++) {
    a[i] += 'y';
  }
}
```

- To update the characters...

Technique #3: Traversing Struct of Arrays

```
void traverse_datavec(int n, DataVec d) {
  char *a = d->cs;
  for (int i = 0; i < n; i++) {
    a[i] += 'y';
  }
}
```

- To update the characters...
- Just iterate over the character...

Technique #3: Traversing Struct of Arrays

```
void traverse_datavec(int n, DataVec d) {
  char *a = d->cs;
  for (int i = 0; i < n; i++) {
    a[i] += 'y';
  }
}
```

- To update the characters...
- Just iterate over the character...
- Higher cache efficiency!

Conclusion

Conclusion

- Memory is hierarchical, with each level slower than predecessors

Conclusion

- Memory is hierarchical, with each level slower than predecessors
- Caching make *locality assumption*

Conclusion

- Memory is hierarchical, with each level slower than predecessors
- Caching make *locality assumption*
- Making this assumption true requires careful design

Conclusion

- Memory is hierarchical, with each level slower than predecessors
- Caching make *locality assumption*
- Making this assumption true requires careful design
- Substantial code alterations can be needed

Conclusion

- Memory is hierarchical, with each level slower than predecessors
- Caching make *locality assumption*
- Making this assumption true requires careful design
- Substantial code alterations can be needed
- But can lead to major performance gains