Programming in C and C++

Lecture 8: The Memory Hierarchy and Cache Optimization

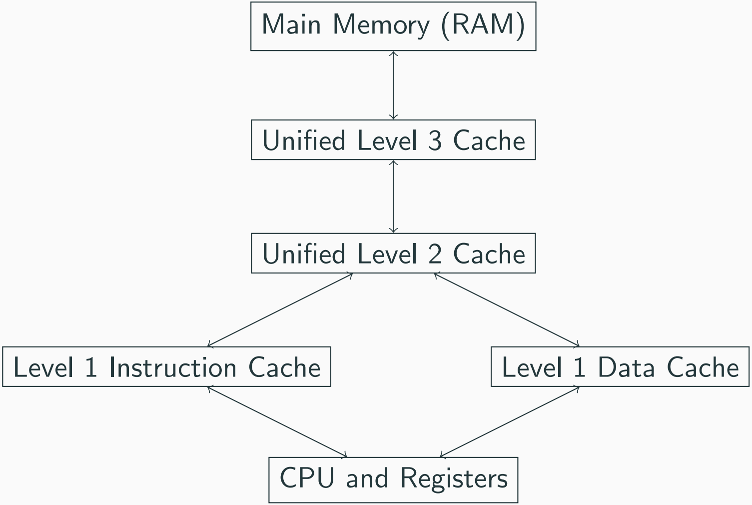
Neel Krishnaswami and Alan Mycroft

Three Simple C Functions

```c
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



Main Memory (RAM)

Unified Level 3 Cache

Unified Level 2 Cache

Level 1 Instruction Cache

Level 1 Data Cache

CPU and Registers

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. . . :

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:

- 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 in C, too

```
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;
}
```

- C uses `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;
}
```

- 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:

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;
}
```

- 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

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

void traverse(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;
};
```

- 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

```
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!

Technique #4: Loop Blocking

```c
#define SIZE 8192
#define dim(i, j) (((i) * SIZE) + (j))

double *add_transpose(double *A,
                      double *B) {
  double *dest =
    malloc(sizeof(double)
           * SIZE * SIZE);
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      dest[dim(i,j)] =
        A[dim(i,j)] + B[dim(j,i)];
    }
  }
  return dest;
}
```

- The `add_transpose` function takes two square matrices A and B, and returns a new matrix equal to $A + B^T$.
- C stores arrays in row-major order.

How Matrices are Laid out in Memory

$$A \triangleq \left\{ \begin{array}{lll} 0 & 1 & 4 \\ 9 & 16 & 25 \\ 36 & 49 & 64 \\ 81 & 100 & 121 \end{array} \right)$$

Address	0	1	2	3	4	5	6	7	8	9	10	11
Value	0	1	4	9	16	25	36	49	64	81	100	121

- A is a 3×4 array.
- $A(i, j)$ is at address $3 \times i + j$ (0 based!)
- E.g., $A(2, 1) = 49$, at address 7
- E.g., $A(3, 1) = 100$, at address 10

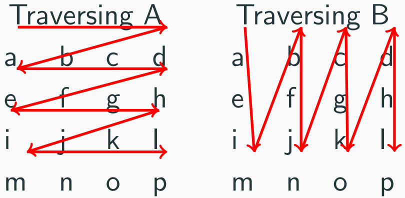
Loop Blocking

```c
#define SIZE 8192
#define dim(i, j) (((i) * SIZE) + (j))

double *add_transpose(double *A,
                      double *B) {
  double *dest =
    malloc(sizeof(double)
           * SIZE * SIZE);
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      dest[dim(i,j)] =
        A[dim(i,j)] + B[dim(j,i)];
    }
  }
  return dest;
}
```

- The succesive accesses to $A(i,j)$ will go sequentially in memory
- The successive accesses to $B(j,i)$ will jump `SIZE` elements at a time

How to Block a Loop, Concept



Traversing A

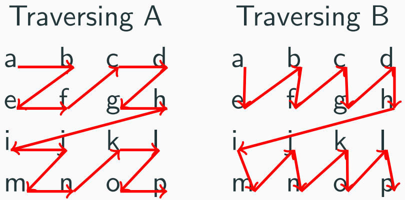
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

Traversing B

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

- We can see that A has a favorable traversal, and B is “jumpy”
- Let’s change the traversal order!

How to Block a Loop, Concept



- Since each nested iteration is acting on the same $n \times n$ submatrix, a cache miss on one lookup will bring memory into cache for the other lookup
- This reduces the total number of cache misses

Loop Blocking

```
double *add_transpose_blocked(double *m1,
                              double *m2,
                              int bsize) {
double *dest =
  malloc(sizeof(double) * SIZE * SIZE);
for (int i = 0; i < SIZE; i += bsize) {
  for (int j = 0; j < SIZE; j += bsize) {
    for (int ii = i; ii < i+bsize; ii++) {
      for (int jj = j; jj < j+bsize; jj++) {
        dest[dim(ii,jj)] =
          m1[dim(ii,jj)] + m2[dim(jj, ii)];
      }
    }
  }
}
return dest;
}
```

- Doubly-nested loop goes to quadruply-nested loop
- Increment `i` and `j` by `bsize` at a time
- Do a little iteration over the submatrix with `ii` and `jj`

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