Performance Improvement

Background reading:
The Practice of Programming (Kernighan & Pike) Chapter 7

"Programming in the Large" Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms (done)
- Modularity (done)
- Building techniques & tools (done)

Debug

- Debugging techniques & tools (done)

Test

- Testing techniques (done)

Maintain

- Performance improvement techniques & tools `<--` we are here

Case study: 25 most common words

Find the 25 most common words in a text file, print their frequencies in decreasing order

$ buzz < novel.txt

```
4503 the
4243 to
3726 of
3654 and
2225 her
2070 i
2012 a
1937 in
1847 was
1710 she
1594 that
1547 it
1450 not
1427 you
1339 he
1271 his
1260 be
1192 as
1177 had
1098 with
1085 for
1007 but
 885 is
 847 have
 800 at
```

Hint 2

No googling for this trivia question:

What work of literature is this?

Hint:
Project Gutenberg's
#1-downloaded book

A program, “buzz.c”

```
/* Enter every word from stdin into a
SymTable, bound to its # of occurrences */
void readInput (SymTable_T table);

/* Make an array of (word, #occ), from
   the contents of the SymTable */
struct counts *extractCounts(
                        SymTable_T table);

/* Sort the “counts” array in descending
order, and print the first 25 entries */
void analyzeData(struct counts *p);

/* The main program */
int main(void) {
   SymTable_T table = SymTable_new();
   readInput(table);
   analyzeData(extractCounts(table));
   return 0;
}
```

Reading the input

```
enum {MAX_LEN = 1000};

int readWord(char *buffer, int buflen) {
  int c;
  /* Skip nonalphabetic characters */
  do {
    c = getchar();
    if (c==EOF) return 0;
  } while (!isalpha(c));
  buffer[0]='\0';
  /* Process alphabetic characters */
  while (isalpha(c)) {
    if (strlen(buffer)<buflen-1) {
       buffer[strlen(buffer)+1]='\0';
       buffer[strlen(buffer)]=tolower(c);
    }
    c=getchar();
  }
  buffer[strlen(buffer)]='\0';
  return 1;
}
```

```
/* Enter every word from stdin into a
SymTable, bound to its # of occurrences */

void readInput (SymTable_T table) {
  char word[MAX_LEN+1];
  while (readWord(word, MAX_LEN+1)) {
   int *p = (int*)SymTable_get(
                                  table, word);
   if (p == NULL) {
         p = (int*)malloc(sizeof(int));
         *p = 0;
         SymTable_put(table, word, p);
   }
   (*p)++;
  }
}
```

Extracting the counts

```
struct word_and_count {
  const char *word;
  int count;
};
struct counts {
  int filled;
  int max;
  struct word_and_count *array;
};

struct counts *makeCounts(int max) {
  struct counts *p =
        (struct counts *) malloc (sizeof (*p));
  assert(p);
  p->filled=0;
  p->max=max;
  p->array = (struct word_and_count *)
     malloc (max * sizeof (struct word_and_count));
  assert (p->array);
  return p;
}
```

```
void handleBinding(
     const char *key,
     void *value, void *extra) {
  struct counts *c = (struct counts *) extra;
  assert (c->filled < c->max);
  c->array[c->filled].word = key;
  c->array[c->filled].count = *((int*)value);
  c->filled += 1;
}

/* Make an array of (word, #occ), from
   the contents of the SymTable */
struct counts *extractCounts(
                              SymTable_T table) {
    struct counts *p = makeCounts(
                      SymTable_getLength(table));
    SymTable_map(table,
                         handleBinding,
                         (void*)p);
    return p;
}
```

Sorting and printing the counts

```
void swap (struct word_and_count *a,
           struct word_and_count *b) {
  struct word_and_count t;
  t=*a; *a=*b; *b=t;
}

void sortCounts (struct counts *counts) {
  /* insertion sort */
  int i,j;
  int n = counts->filled;
  struct word_and_count *a = counts->array;
  for (i=1; i<n; i++) {
    for (j=i;
         j>0 && a[j-1].count<a[j].count;
         j--)
      swap(a+j, a+j-1);
  }
}
```

```
/* Sort the "counts" array in descending
order, and print the first 25 entries */

void analyzeData(struct counts *p) {
    int i, n;
    assert (p->filled == p->max);
    sortCounts(p);
    n = 25<p->max ? 25 : p->max;
    for (i=0; i<n; i++)
      printf("%10d %s\n",
             p->array[i].count,
             p->array[i].word);
}
```

Timing a Program

Run a tool to time program execution

- E.g., Unix **`time`** command

```
$ time ./buzz < corpus.txt > output.txt

3.58user 0.00system 0:03.59elapsed 99%CPU
```

Output:

- **Real (or "elapsed")**: Wall-clock time between program invocation and termination
- **User**: CPU time spent executing the program
- **System**: CPU time spent within the OS on the program's behalf

In summary: takes 3.58 seconds to process 703,549 characters of input. That's really slow!

(especially if we want to process a whole library of books)

What should you do?

The COS 226 answer:

Use asymptotically efficient algorithms and data structures everywhere.

WRONG!

*(and, to be fair, that was a **caricature** of the COS 226 answer)*

What should you do?

Caricature of the COS 226 answer:

Use asymptotically efficient algorithms and data structures everywhere.

Most parts of your program won't run on "big data!" Simplicity, maintainability, correctness, easy algorithms and data structures are most important.

Words of the sages

"Optimization hinders evolution."

-- Alan Perlis

"Premature optimization is the root of all evil."

-- Donald Knuth

"Rules of Optimization:

- Rule 1: Don't do it.
- Rule 2 (for experts only): Don't do it yet."

-- Michael A. Jackson*

*The MIT professor, not the pop singer.

When to Improve Performance

"The first principle of optimization is

<u>don't</u>.

Is the program good enough already?
Knowing how a program will be used
and the environment it runs in,
is there any benefit to making it faster?"

-- Kernighan & Pike

When to Improve Performance

"The first principle of optimization is

The only reason we're even allowed to be here (as good software engineers) is because we did the performance measurement (700k characters in 3.58 seconds) and found it unacceptable.

is there any benefit to making it faster?"

-- Kernighan & Pike

Goals of this Lecture

Help you learn about:

- Techniques for improving program performance
  - How to make your programs run faster and/or use less memory
- The **oprofile** execution profiler

Why?

- In a large program, typically a small fragment of the code consumes most of the CPU time and/or memory
- A good software engineer knows how to identify such fragments,
- and knows how to improving their performance

Performance Improvement Pros

Techniques described in this lecture can yield answers to questions such as:

- How slow is my program?
- Where is my program slow?
- Why is my program slow?
- How can I make my program run faster?
- How can I make my program use less memory?

Timing Parts of a Program

Call a function to compute **wall-clock time** consumed

- E.g., Unix **`gettimeofday()`** function (time since Jan 1, 1970)

```
#include <sys/time.h>

struct timeval startTime;
struct timeval endTime;
double wallClockSecondsConsumed;

gettimeofday(&startTime, NULL);
<execute some code here>
gettimeofday(&endTime, NULL);
wallClockSecondsConsumed =
   endTime.tv_sec - startTime.tv_sec +
   1.0E-6 * (endTime.tv_usec - startTime.tv_usec);
```

Timing Parts of a Program (cont.)

Call a function to compute **CPU time** consumed

- E.g. **`clock()`** function

```
#include <time.h>

clock_t startClock;
clock_t endClock;
double cpuSecondsConsumed;

startClock = clock();
<execute some code here>
endClock = clock();
cpuSecondsConsumed =
   ((double)(endClock - startClock)) / CLOCKS_PER_SEC;
```

Identifying Hot Spots

Gather statistics about your program's execution

- How much time did execution of a particular function take?
- How many times was a particular function called?
- How many times was a particular line of code executed?
- Which lines of code used the most time?
- Etc.

How? Use an **execution profiler**

- Example: **`gprof`** (GNU Performance Profiler)

- Reports how many seconds spent in each of your programs' functions, to the nearest millisecond.

Identifying Hot Spots

Gather statistics a

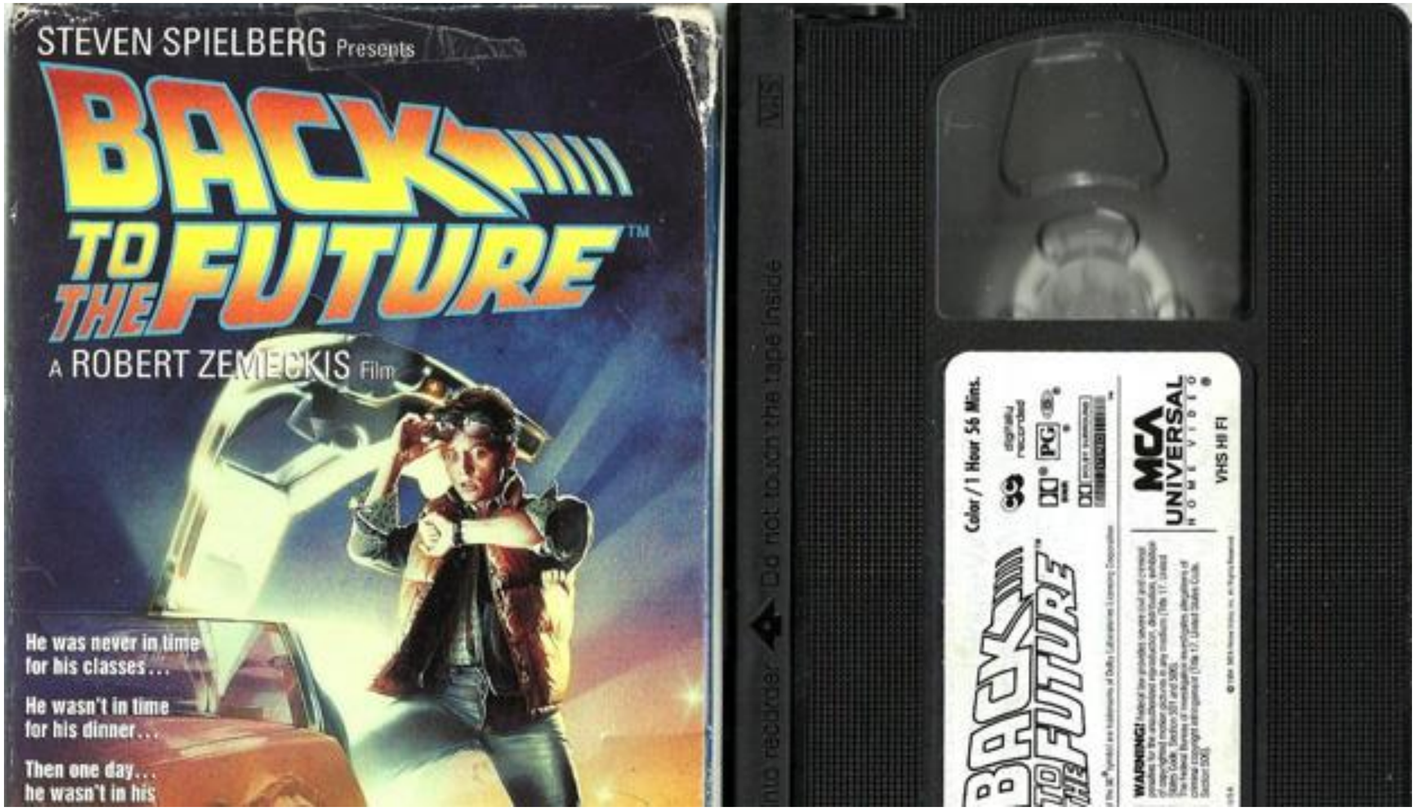
- How much
- How ma
- How ma
- Which line
- Etc.

Milliseconds? Really?
My whole program runs in a couple of milliseconds!
What century do you think we're in?

How? Use an **execution pr**

- Example: **`gprof`** (GNU Perform Profiler)
- Reports how many seconds spen in each of your programs' functions, to the nearest millisecond.

The 1980s just called, they want their profiler back . . .



For some reason, between 1982 and 2016 while computers got 1000x faster, nobody thought to tweak **gprof** to make it report to the nearest microsecond instead of millisecond.

The 1980s just called, they want their profiler back . . .

So we will use **oprofile**, a 21st-century profiling tool.
But **gprof** is still available and convenient:
what I show here (with **oprofile**) can be done with **gprof**.

Read the man pages:

```
$ man gprof
$ man oprofile
```

Using `oprofile`

Step 1: Compile the program with –g and –O2

```
gcc -g -O2 -c buzz.c; gcc buzz.o symtablelist.o -o buzz1
```

-g adds “symbol table” to buzz.o (and the eventual executable)

-O2 says “compile with optimizations.” If you’re worried enough about performance to want to profile, then measure the compiled-for-speed version of the program.

Step 2: Run the program

```
operf ./buzz1 < corpus.txt >output
```

- Creates subdirectory **oprofile_data** containing statistics

Step 3: Create a report

```
opreport -l -t 1 > myreport
```

- Uses **oprofile_data** and **buzz**’s symbol table to create textual report

Step 4: Examine the report

```
cat myreport
```

The oprofile report

samples	%	image name	app name	symbol name
20871	75.8807	[illegible]	buzz1	__strcmp_sse42
5732	20.8398	buzz1	buzz1	SymTable_get
257	0.9344	buzz1	buzz1	SymTable_put
256	0.9307	buzz1	buzz1	[illegible]ortCounts
105	0.3817	buzz1	buzz1	read[illegible]rd
92	0.3345	no-vmlinux	buzz1	/no-vmlin[illegible]
75	0.2727	libc-2.17.so	buzz1	fgetc
73	0.2654	libc-2.17.so	buzz1	__strlen_sse2_pmi[illegible]
[illegible]	0.0364	buzz1	buzz1	readInput
[illegible]	0.0327	libc-2.17.so	buzz1	__ctype_tolower_lo[illegible]
[illegible]	[illegible]	libc-2.17.so	buzz1	_int_malloc
[illegible]	[illegible].0109	libc-2.17.so	buzz1	__ctype_b_loc
[illegible]	0.0109	libc-2.17.so	buzz1	malloc
[illegible]	0.0073	libc-2.17.so	buzz1	__strcpy_sse2_unaligned
1	0.0036	buzz1	buzz1	SymTable_map
1	0.0036	ld-2.17.so	time	bsearch
1	0.0036	libc-2.17.so	buzz1	malloc_consolidate
1	0.0036	libc-2.17.so	buzz1	strcpy
1	0.0036	libc-2.17.so	time	__write_nocancel

% of execution time spent in this function

Name of the function

Shell command that ran this

Name of the binary executable

I've left out the `-t 1` here; otherwise it would leave out any line whose % is less than 1

What do we learn from this?

samples	%	image name	app name	symbol name
20871	75.8807	libc-2.17.so	buzz1	__strcmp_sse42
5732	20.8398	buzz1	buzz1	SymTable_get
257	0.934[illegible]	buzz1	buzz1	SymTable_put
256	0.9307	buzz1	buzz1	sortCounts
105	0.3817	[illegible]	buzz1	readWord
92	0.3345	[illegible]	buzz1	/no-vmlinux
75	0.2727	[illegible]	buzz1	fgetc
73	0.2654	[illegible]	[illegible]	__strlen_sse2_pminub
10	0.0364	[illegible]	[illegible]	readInput
9	0.0327	[illegible]	[illegible]	__ctype_tolower_loc
8	0.0291	[illegible]	[illegible]	_int_malloc
3	0.0109	[illegible]	[illegible]	__ctype_b_loc
3	0.0109	[illegible]	[illegible]	malloc
2	0.0073	[illegible]	[illegible]	__strcpy_sse2_unaligned
1	0.0036	buzz1	buzz1	SymTable_map
1	0.0036	ld-2.17.so	time	bsearch
1	0.0036	libc-2.17.so	buzz1	malloc_consolidate
1	0.0036	libc-2.17.so	buzz1	strcpy
1	0.0036	libc-2.17.so	time	__write_nocancel

96% of execution time is in strcmp( ) and in SymTable_get( )

Who is calling strcmp? Nothing in buzz.c . . .

It's the symtablelist.c implementation of SymTable_get . . .

Use better algorithms and data structures

Improve the "buzz" program by using

symtablehash.c instead of symtablelist.c

```
gcc -g -O2 -c buzz.c; gcc buzz.o symtablelist.o -o buzz1

gcc -g -O2 -c buzz.c; gcc buzz.o symtablehash.o -o buzz2
```

Result: execution time decreases from

3.58 seconds to 0.06 seconds

The use of insertion sort instead of quicksort doesn't actually seem to be a problem! That's what we learned from doing the **oprofile**. *This is engineering, not just hacking.*

What if 0.06 seconds isn't fast enough?

```
operf ./buzz2 < corpus.txt >output
opreport -l -t 1 > myreport
```

```
samples  %        image name     app name   symbol name
221      39.6057  buzz2          buzz2      sortCounts
66       11.8280  buzz2          buzz2      SymTable_get
66       11.8280  libc-2.17.so   buzz2      __strlen_sse2_pminub
50        8.9606  buzz2          buzz2      SymTable_hash
45        8.0645  libc-2.17.so   buzz2      fgetc
37        6.6308  buzz2          buzz2      readWord
20        3.5842  libc-2.17.so   buzz2      __strcmp_sse42
20        3.5842  no-vmlinux     buzz2      /no-vmlinux
```

40% of execution time in sortCounts. Let's make it faster.

Line-by-line view in oprofile

```
operf ./buzz2 <corpus.txt >output2
opannotate -s > annotated-source2
```

The file `annotated-source2`:

```
               :/*----------- Sort the counts  ----------------*/
               :
               :void swap (struct word_and_count *a,
               :           struct word_and_count *b) {
               :  struct word_and_count t;
  87   21.42   :  t=*a; *a=*b; *b=t;
               :}
               :
               :void sortCounts (struct counts *counts) {
               :  /* insertion sort */
               :  int i,j;
               :  int n = counts->filled;
               :  struct word_and_count *a = counts->array;
               :  for (i=1; i<n; i++) {
  81   19.95   :    for (j=i; j>0 && a[j-1].count<a[j].count; j--)
               :      swap(a+j, a+j-1);
               :  }
               :}
```

samples

percentage

source lines

Insertion Sort

```
void swap (struct word_and_count *a,
           struct word_and_count *b) {
  struct word_and_count t;
  t=*a; *a=*b; *b=t;
}

void sortCounts (struct counts *counts) {
  /* insertion sort */
  int i,j;
  int n = counts->filled;
  struct word_and_count *a = counts->array;
  for (i=1; i<n; i++) {
    for (j=i;
         j>0 && a[j-1].count<a[j].count;
         j--)
      swap(a+j, a+j-1);
  }
}
```

Quicksort

Use the **`qsort`** function
from the standard library
(covered in precept last week)

```
int compare_count(
        const void *p, const void *q) {
 return
   ((struct word_and_count *)q)->count
  – ((struct word_and_count *)p)->count;
}


void sortCounts (struct counts *counts) {
  qsort(counts->array,
        counts->filled,
        sizeof(struct word_and_count),
        compare_count);
}
```

Use quicksort instead of insertion sort

Result: execution time decreases from

0.06 seconds to 0.04 seconds

We could have predicted this! If 40% of the time was in the sort function, and we practically eliminate all of that, then it'll be 40% faster.

Is that fast enough? Well, yes.

But just for fun, let's run the profiler again.

What if 0.04 seconds isn't fast enough?

samples	%	image name	app name	symbol name
73	27.3408	libc-2.17.so	buzz3	__strlen_sse2_pminub
48	17.9775	buzz3	buzz3	readWord
36	13.4831	buzz3	buzz3	SymTable_hash
33	12.3596	libc-2.17.so	buzz3	fgetc
27	10.1124	buzz3	buzz3	SymTable_get
15	5.6180	no-vmlinux	buzz3	/no-vmlinux
11	4.1199	libc-2.17.so	buzz3	__strcmp_sse42
4	1.4981	libc-2.17.so	buzz3	_int_malloc
3	1.1236	libc-2.17.so	buzz3	msort_with_t

27% of execution time in strlen( ). Who's calling strlen( ) ?

Reading the input

```
enum {MAX_LEN = 1000};

int readWord(char *buffer, int buflen) {
  int c;
  /* Skip nonalphabetic characters */
  do {
    c = getchar();
    if (c==EOF) return 0;
  } while (!isalpha(c));
  buffer[0]='\0';
  /* Process alphabetic ch
  while (isalpha(c)) {
    if (strlen(buffer)<b   len-1) {
      buffer[strlen(buffer)+1]='\0';
      buffer[strlen(buffer)]=t     er(c);
    }
    c=getchar();
  }
  buffer[strlen(buffer)]='\0';
  return 1;
}
```

This is just silly. We could keep track of the length of the buffer in an integer variable, instead of recomputing each time.

How much faster would the program become?

27% faster; from 0.04 sec to 0.03 sec.

Is it worth it? Perhaps, especially if the program doesn't become harder to read and maintain.

Enabling Speed Optimization

Enable compiler speed optimization

```
gcc217 -Ox mysort.c -o mysort
```

- Compiler spends more time compiling your code so…
- Your code spends less time executing
- `x` can be:
  - **0:** don't optimize
  - **1**: optimize (this is the default)
  - **2**: optimize more
  - **3**: optimize across .c files
- See "man gcc" for details

Beware: Speed optimization can affect debugging

e.g. Optimization eliminates variable ⇒ GDB cannot print value of variable

Agenda

Execution (time) efficiency

- Do timing studies
- Identify hot spots
- Use a better algorithm or data structure
- Enable compiler speed optimization
- **Tune the code**

Memory (space) efficiency

Agenda

Execution (time) efficiency

- Do timing studies
- Identify hot spots
- Use a better algorithm or data structure
- Enable compiler speed optimization
- **Tune the code**

Warning! Much of what I'm about to show you will be done *automatically* by the C compiler: gcc –O3 *so you usually shouldn't do it yourself!*

Avoiding Repeated Computation

Avoid repeated computation

Before:

```
int g(int x)
{   return f(x) + f(x) + f(x) + f(x);
}
```

After:

```
int g(int x)
{   return 4 * f(x);
}
```

Could a good compiler do that for you?

Aside: Side Effects as Blockers

```
int g(int x)
{   return f(x) + f(x) + f(x) + f(x);
}
```

```
int g(int x)
{   return 4 * f(x);
}
```

Q: Could a good compiler do that for you?

A: Probably not

Suppose `f()` has **side effects**?

```
int counter = 0;
...
int f(int x)
{   return counter++;
}
```

And `f()` might be defined in another file known only at link time!

Avoiding Repeated Computation

Avoid repeated computation

Before:

```
for (i = 0; i < strlen(s); i++)
{  /* Do something with s[i] */
}
```

After:

```
length = strlen(s);
for (i = 0; i < length; i++)
{  /* Do something with s[i] */
}
```

Could a good compiler do that for you?

Avoiding Repeated Computation

Avoid repeated computation

Before:

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      a[n*i + j] = b[j];
```

After:

```
for (i = 0; i < n; i++)
{  ni = n * i;
   for (j = 0; j < n; j++)
      a[ni + j] = b[j];
}
```

Could a good compiler do that for you?

Tune the Code

Avoid repeated computation

Before:

```
void twiddle(int *p1, int *p2)
{   *p1 += *p2;
    *p1 += *p2;
}
```

After:

```
void twiddle(int *p1, int *p2)
{   *p1 += *p2 * 2;
}
```

Could a good compiler do that for you?

Aside: Aliases as Blockers

```
void twiddle(int *p1, int *p2)
{  *p1 += *p2;
   *p1 += *p2;
}
```

```
void twiddle(int *p1, int *p2)
{  *p1 += *p2 * 2;
}
```

Q: Could a good compiler do that for you?

A: Not necessarily

What if `p1` and `p2` are **aliases**?

- What if `p1` and `p2` point to the same integer?
- First version: result is 4 times `*p1`
- Second version: result is 3 times `*p1`

Some compilers support `restrict` keyword

Inlining Function Calls

Inline function calls

Could a good compiler do that for you?

Before:

```
void g(void)
{  /* Some code */
}
void f(void)
{  …
   g();
   …
}
```

After:

```
void f(void)
{  …
   /* Some code */
   …
}
```

Beware: Can introduce redundant/cloned code
Some compilers support `inline` keyword

Unrolling Loops

Could a good compiler do that for you?

Unroll loops

Original:

```
for (i = 0; i < 6; i++)
   a[i] = b[i] + c[i];
```

Maybe faster:

```
for (i = 0; i < 6; i += 2)
{  a[i+0] = b[i+0] + c[i+0];
   a[i+1] = b[i+1] + c[i+1];
}
```

Maybe even faster:

```
a[i+0] = b[i+0] + c[i+0];
a[i+1] = b[i+1] + c[i+1];
a[i+2] = b[i+2] + c[i+2];
a[i+3] = b[i+3] + c[i+3];
a[i+4] = b[i+4] + c[i+4];
a[i+5] = b[i+5] + c[i+5];
```

Some compilers provide option, e.g. `-funroll-loops`

Using a Lower-Level Language

Rewrite code in a lower-level language

- As described in second half of course…
- Compose key functions in **assembly language** instead of C
  - Use registers instead of memory
  - Use instructions (e.g. **`adc`**) that compiler doesn’t know

Beware: Modern optimizing compilers generate fast code

- Hand-written assembly language code could be slower!

Could a good compiler do that for you?

Clarity supersedes performance

Don’t improve performance unless you must!!!