Profiling and Optimising C
Monash/NCI Training Week

Stephen Sanderson

National Computational Infrastructure, Australia

Acknowledgement of Country

The National Computational Infrastructure acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Outline

In this session, we will cover:

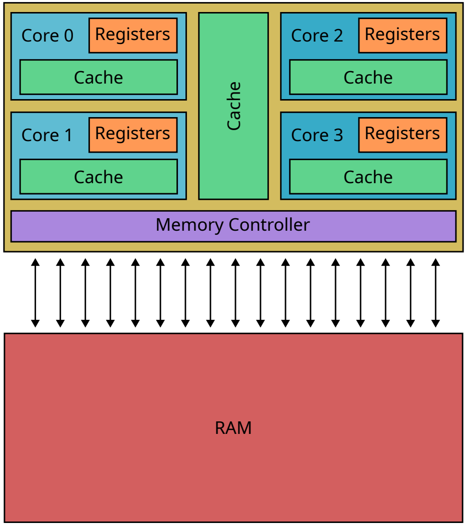
- Understanding computer architecture to guide optimisation
- General approaches to optimising code
- Tools for profiling code to find optimisation targets
- Some common performance considerations

Computer architecture

Computer Architecture

To have some intuition about how to write fast code, it's beneficial to understand the basics of how a computer works.

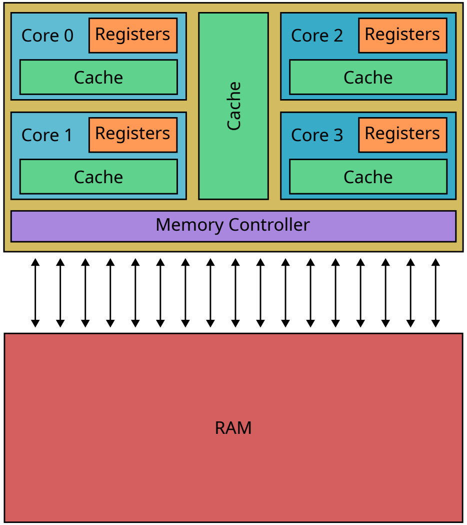
- For speed: Registers > Local Cache > Shared Cache > RAM



Computer Architecture

To have some intuition about how to write fast code, it's beneficial to understand the basics of how a computer works.

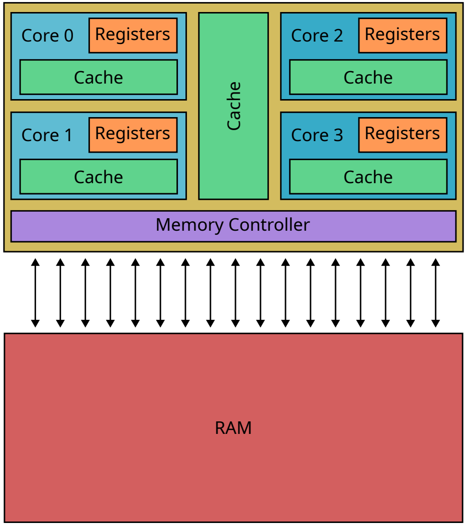
- For speed: Registers > Local Cache > Shared Cache > RAM
- Chunks of memory aligned to the size of a *cache line* (typically 64 bytes) are fetched from RAM in one go.



Computer Architecture

To have some intuition about how to write fast code, it's beneficial to understand the basics of how a computer works.

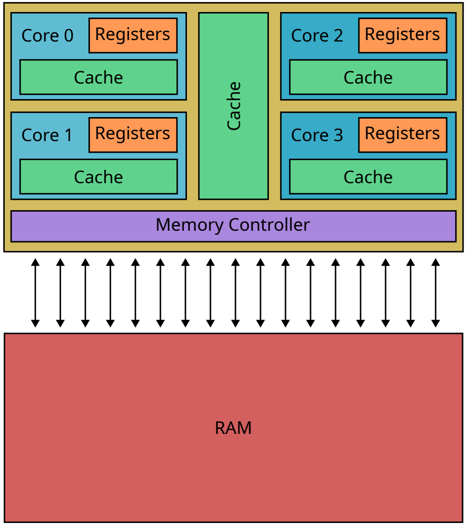
- For speed: Registers > Local Cache > Shared Cache > RAM
- Chunks of memory aligned to the size of a *cache line* (typically 64 bytes) are fetched from RAM in one go.
- Data is loaded into registers, instructions are executed to manipulate registers, then results are stored back into memory.



Computer Architecture

To have some intuition about how to write fast code, it's beneficial to understand the basics of how a computer works.

- For speed: Registers > Local Cache > Shared Cache > RAM
- Chunks of memory aligned to the size of a *cache line* (typically 64 bytes) are fetched from RAM in one go.
- Data is loaded into registers, instructions are executed to manipulate registers, then results are stored back into memory.
- CPUs are smart! They will try to execute multiple instructions at once and predict branching logic.



Compilers

Compilers are also smart!
Turning on compiler optimisations can enable things like:

- Inlining
- Loop unrolling (or partial unrolling)
- Loop lifting
- Tail call optimisations
- Auto-vectorisation (but probably not as often as you expect!)
- Much more

Inlining

```
void quad_roots(double a, double b, double c,
                double* root1, double* root2)
{
    *root1 = (-b + sqrt(b * b - 4. * a * c)) / (2. * a);
    *root2 = (-b - sqrt(b * b - 4. * a * c)) / (2. * a);
}

double largest_root(double a, double b, double c) {
    double root1, root2;
    quad_roots(a, b, c, &root1, &root2);
    if fabs(root1) > fabs(root2) {
        return root1;
    } else {
        return root2;
    }
}
```

Inlining

```
void quad_roots(double a, double b, double c,
                double* root1, double* root2)
{
    *root1 = (-b + sqrt(b * b - 4. * a * c)) / (2. * a);
    *root2 = (-b - sqrt(b * b - 4. * a * c)) / (2. * a);
}

double largest_root(double a, double b, double c) {
    double root1, root2;
    quad_roots(a, b, c, &root1, &root2);
    if fabs(root1) > fabs(root2) {
        return root1;
    } else {
        return root2;
    }
}
```

$\rightarrow$

```
double largest_root(double a, double b, double c) {
    double root1 = (-b + sqrt(b*b - 4.*a*c)) / (2.*a);
    double root2 = (-b - sqrt(b*b - 4.*a*c)) / (2.*a);
    if fabs(root1) > fabs(root2) {
        return root1;
    } else {
        return root2;
    }
}
```

Inlining

```c
// We can encourage (but not guarantee) inlining with
// the `inline` keyword
inline void quad_roots(double a, double b, double c,
                       double* root1, double* root2)
{
    *root1 = (-b + sqrt(b * b - 4. * a * c)) / (2. * a);
    *root2 = (-b - sqrt(b * b - 4. * a * c)) / (2. * a);
}

double largest_root(double a, double b, double c) {
    double root1, root2;
    quad_roots(a, b, c, &root1, &root2);
    if fabs(root1) > fabs(root2) {
        return root1;
    } else {
        return root2;
    }
}
```

$\rightarrow$

```c
double largest_root(double a, double b, double c) {
    double root1 = (-b + sqrt(b*b - 4.*a*c)) / (2.*a);
    double root2 = (-b - sqrt(b*b - 4.*a*c)) / (2.*a);
    if fabs(root1) > fabs(root2) {
        return root1;
    } else {
        return root2;
    }
}
```

Inlining

```c
// We can encourage (but not guarantee) inlining with
// the `inline` keyword
inline void quad_roots(double a, double b, double c,
                       double* root1, double* root2)
{
    *root1 = (-b + sqrt(b * b - 4. * a * c)) / (2. * a);
    *root2 = (-b - sqrt(b * b - 4. * a * c)) / (2. * a);
}

double largest_root(double a, double b, double c) {
    double root1, root2;
    quad_roots(a, b, c, &root1, &root2);
    if fabs(root1) > fabs(root2) {
        return root1;
    } else {
        return root2;
    }
}
```

$\rightarrow$

```c
double largest_root(double a, double b, double c) {
    double root1 = (-b + sqrt(b*b - 4.*a*c)) / (2.*a);
    double root2 = (-b - sqrt(b*b - 4.*a*c)) / (2.*a);
    if fabs(root1) > fabs(root2) {
        return root1;
    } else {
        return root2;
    }
}
```

For functions that are `#include`d from a header file, the most reliable way to get them inlined is to declare them as `static inline` and write the full implementation in the header.

Loop unrolling

```
for (int i = 0; i < 10; ++i) {
    list_append(&list, i*i);
}
```

Loop unrolling

```
for (int i = 0; i < 10; ++i) {
    list_append(&list, i*i);
}
```

$\rightarrow$

```
list_append(&list, 0);
list_append(&list, 1);
list_append(&list, 4);
list_append(&list, 9);
list_append(&list, 16);
list_append(&list, 25);
list_append(&list, 36);
list_append(&list, 49);
list_append(&list, 64);
list_append(&list, 81);
```

Loop lifting

```c
for (int i = 0; i < 10; ++i) {
  if (store_negatives) {
    list_append(&list, -i*i);
  } else {
    list_append(&list, i*i);
  }
}
```

Loop lifting

```
for (int i = 0; i < 10; ++i) {
  if (store_negatives) {
    list_append(&list, -i*i);
  } else {
    list_append(&list, i*i);
  }
}
```

$\rightarrow$

```
// store_negatives can't change during
// the loop, so only check it once
if (store_negatives) {
  for (int i = 0; i < 10; ++i) {
    list_append(&list, -i*i);
  }
else {
  for (int i = 0; i < 10; ++i) {
    list_append(&list, i*i);
  }
}
```

Tail call optimisations

If the final instruction of a function (e.g. `foo`) is to call another function (e.g. `bar`), the compiler can overwrite `foo`'s stack frame when `bar` is called instead of preserving it.

```
double foo() {
    // do things
    return bar(a, b);
}
```

This can be especially useful for recursive functions.

Tail call optimisations

If `foo` needs to do things with the result of `bar`, then this optimisation may not be possible:

```
double foo() {
    // do things
    return c * bar(a, b);
}
```

Auto-vectorisation

Vector registers can hold a number of values packed together, and operate on them simultaneously. For example, AVX2 (supported by most modern x86 CPUs) has 256 bit vector registers, allowing up to 4 `double`s or 8 `float`s to be operated on at once.

```
for (int i = 0; i < N; ++i)
    y[i] = a*x[i]*x[i] + b*x[i] + c;
```

Auto-vectorisation

Vector registers can hold a number of values packed together, and operate on them simultaneously. For example, AVX2 (supported by most modern x86 CPUs) has 256 bit vector registers, allowing up to 4 `double`s or 8 `float`s to be operated on at once.

```
for (int i = 0; i < N; ++i)
    y[i] = a*x[i]*x[i] + b*x[i] + c;
```

$\downarrow$

```
int i = 0;
for (; i + 3 < N; i += 4) {
    y[i]   = a*x[i] * x[i]    + b*x[i]    + c;
    y[i+1] = a*x[i+1]*x[i+1] + b*x[i+1] + c;
    y[i+2] = a*x[i+2]*x[i+2] + b*x[i+2] + c;
    y[i+3] = a*x[i+3]*x[i+3] + b*x[i+3] + c;
}
for (; i < N; ++i)
    y[i] = a*x[i]*x[i] + b*x[i] + c;
```

Compilers

Compilers are generally good at finding these optimisations, but they must abide by the “as if" rule. That is, the transformations they make must produce the same observable result as if the code you wrote was run.

Compilers

Compilers are generally good at finding these optimisations, but they must abide by the “as if" rule. That is, the transformations they make must produce the same observable result as if the code you wrote was run.

Because of this, one thing compilers won’t do for you is optimise your memory layout.

Compilers

Compilers are generally good at finding these optimisations, but they must abide by the “as if" rule. That is, the transformations they make must produce the same observable result as if the code you wrote was run.

Because of this, one thing compilers won’t do for you is optimise your memory layout.

They will also be conservative in making some transformations if they can’t prove there won’t be an observable difference. This can become tricky when pointers are involved if the compiler isn’t sure whether the memory pointed to overlaps (i.e. the pointers *alias*).

Compilers

Compilers are generally good at finding these optimisations, but they must abide by the “as if" rule. That is, the transformations they make must produce the same observable result as if the code you wrote was run.

Because of this, one thing compilers won’t do for you is optimise your memory layout.

They will also be conservative in making some transformations if they can’t prove there won’t be an observable difference. This can become tricky when pointers are involved if the compiler isn’t sure whether the memory pointed to overlaps (i.e. the pointers *alias*).

When optimising code, it pays to be aware of what a compiler can and can’t do for you, and try to help it along where possible.

General approach to optimisation

Optimising

The simplest method for speeding up code is profile-guided optimisation:

1. Get initial implementation working
2. Run with a profiler
3. Identify "hot spots" (i.e. code that takes up the most time)
4. Apply appropriate optimisation techniques to the hot spots
   - Vectorisation
   - Parallelisation
   - GPU acceleration (or use of other accelerators)
   - Algorithmic optimisation
   - Machine-specific optimisation (e.g. designing algorithms with the CPU's cache size in mind, or using inline assembly to take advantage of particular CPU properties)
5. Check for correctness (it helps to have a comprehensive set of unit and integration tests to verify changes)
6. Repeat from step 2

Optimising

But what if there are no hot spots and the code is still slow?

- This can be harder and/or more time-consuming to deal with, since it often requires changes to the architecture of the program.

Optimising

But what if there are no hot spots and the code is still slow?

- This can be harder and/or more time-consuming to deal with, since it often requires changes to the architecture of the program.
- Intuition, experience, and a good understanding of your program all help here.

Optimising

But what if there are no hot spots and the code is still slow?

- This can be harder and/or more time-consuming to deal with, since it often requires changes to the architecture of the program.
- Intuition, experience, and a good understanding of your program all help here.
- Think about what your program is trying to achieve, how it does so, and whether it can achieve the same result with less "work" or less overhead.

Optimising

But what if there are no hot spots and the code is still slow?

- This can be harder and/or more time-consuming to deal with, since it often requires changes to the architecture of the program.
- Intuition, experience, and a good understanding of your program all help here.
- Think about what your program is trying to achieve, how it does so, and whether it can achieve the same result with less "work" or less overhead.
- "Work" can include things like:
  - Loading from/storing to memory (particularly heap memory).
  - Performing a small calculation – if it's repeated in a few functions, you could consider storing the result. Conversely, if the result is needed by a few parallel threads, it might be faster to have each thread perform its own calculation to avoid waiting on a lock.

Optimising

But what if there are no hot spots and the code is still slow?

- This can be harder and/or more time-consuming to deal with, since it often requires changes to the architecture of the program.
- Intuition, experience, and a good understanding of your program all help here.
- Think about what your program is trying to achieve, how it does so, and whether it can achieve the same result with less "work" or less overhead.
- "Work" can include things like:
  - Loading from/storing to memory (particularly heap memory).
  - Performing a small calculation – if it's repeated in a few functions, you could consider storing the result. Conversely, if the result is needed by a few parallel threads, it might be faster to have each thread perform its own calculation to avoid waiting on a lock.
- Overhead might include things like memory allocation, cache misses, or spawning threads, or sometimes even small things like function calls.

Optimising

But what if there are no hot spots and the code is still slow?

- This can be harder and/or more time-consuming to deal with, since it often requires changes to the architecture of the program.
- Intuition, experience, and a good understanding of your program all help here.
- Think about what your program is trying to achieve, how it does so, and whether it can achieve the same result with less "work" or less overhead.
- "Work" can include things like:
  - Loading from/storing to memory (particularly heap memory).
  - Performing a small calculation – if it's repeated in a few functions, you could consider storing the result. Conversely, if the result is needed by a few parallel threads, it might be faster to have each thread perform its own calculation to avoid waiting on a lock.
- Overhead might include things like memory allocation, cache misses, or spawning threads, or sometimes even small things like function calls.
- With experience, you can learn to design software in a way that avoids overhead where possible. Don't be afraid to re-write once you have a better understanding of the problem.

Optimising

Example:

- You have a `struct` which contains many members, and you're storing many instances of it in an array.
- You have various sections of code which each iterate over the array and operate on a small subset of the `struct` members.

Optimising

Example:

- You have a `struct` which contains many members, and you're storing many instances of it in an array.
- You have various sections of code which each iterate over the array and operate on a small subset of the `struct` members.

Possible optimisation strategy:

- Instead of a `struct` of concrete values, make it store pointers instead, and allocate a separate array for each member.
- This way, iterating over each member will only fetch necessary data for better cache behaviour.
- An alternative strategy would be to perform all required operations on one instance of the original `struct` before moving on to the next, but this probably won't be as fast since modern CPUs tend to prefer performing the same small set of operations repeatedly on contiguous blocks of memory.

Optimising

What it looks like:

```
typedef struct {
    double a;
    double b;
    double c;
    int N;
    // many more ...
} MyData;
MyData* data_array = malloc(sizeof(MyData) * MANY);

void do_something(MyData* array, const size_t MANY) {
    for (size_t i = 0; i < MANY; ++i) // Only need a, b, c, but reads all
        array[i].c = 2. * array[i].a + array[i].b;
}
```

Optimising

What it looks like:

```
typedef struct {
    double* a;
    double* b;
    double* c;
    int* N;
    // many more ...
} MyData;
MyData data_array = { .a = malloc(sizeof(double) * MANY),
                      .b = malloc(sizeof(double) * MANY),
                      .c = malloc(sizeof(double) * MANY),
                      .N = malloc(sizeof(int) * MANY),
                      // ...
};
void do_something(MyData array, const size_t MANY) {
    for (size_t i = 0; i < MANY; ++i) // Here we only read a, b & c
        array.c[i] = 2. * array.a[i] + array.b[i];
}
```

Profiling tools

Profiling Tools

There are two main models of profiling methods:

Sampling:
Periodically stop the program and use information like the program counter and stack trace to calculate an approximation of how much time is spent in each function or on each line.

Instrumentation:
Either automatically or manually insert extra instructions to time the execution of a block of code.

Generally want to compile code with optimisations enabled when profiling, but keep the `-g` flag for debug symbols so that the profiler knows which parts of the executable correspond to which parts of the source code.

A selection of profiling tools

`gprof`

- Open source profiler for Linux.
- Uses compile-time instrumentation to count function calls.
- Samples the program many times during execution to see where the program counter is to approximate time spent in each function.

A selection of profiling tools

`gprof`

- Open source profiler for Linux.
- Uses compile-time instrumentation to count function calls.
- Samples the program many times during execution to see where the program counter is to approximate time spent in each function.

`callgrind`

- A tool within `valgrind`
- Gives deterministic, reproducible profiles by counting the number of instructions executed for each line of source code.

A selection of profiling tools

`gprof`

- Open source profiler for Linux.
- Uses compile-time instrumentation to count function calls.
- Samples the program many times during execution to see where the program counter is to approximate time spent in each function.

`callgrind`

- A tool within `valgrind`
- Gives deterministic, reproducible profiles by counting the number of instructions executed for each line of source code.

Arm MAP

- Proprietary profiler for HPC
- Works well with MPI, OpenMP, etc.
- Graphical interface

Some hands-on examples

Appending to `List`

Let's re-visit the `List` code from the previous sessions.

How does it perform if we append to a `List` many times?

Follow through the notebook to profile this with `gprof` and `callgrind` and see what optimisations you can come up with.

Appending to `List`

Re-allocating memory every time an element is appended is slow!

We can speed this up by additionally storing a `capacity` value, so we can grow the list by a large amount with a `list_reserve` function, and only re-allocate when we're about to exceed the capacity.

This could be extended (and often is) to over-allocate whenever `list_reserve` is called. A common strategy is to reserve capacity up to the next power of two.

Arm MAP

`gprof` and `callgrind` are open source and broadly available, but they have their limitations.

Another useful profiling tool available on Gadi is Arm MAP (part of Arm Forge). It has a graphical interface, and importantly it has good support for MPI and OpenMP acceleration.

It can be used either by running your program directly through the Arm MAP GUI, or you can use `map` from the command line with the `--profile` flag to generate a profile file which can be opened and analysed later (useful for large or long-running jobs):

```
module load arm-forge/22.1.1
map --profile mpirun my_mpi_app
```

Arm MAP

`gprof` and `callgrind` are open source and broadly available, but they have their limitations.

Another useful profiling tool available on Gadi is Arm MAP (part of Arm Forge). It has a graphical interface, and importantly it has good support for MPI and OpenMP acceleration.

It can be used either by running your program directly through the Arm MAP GUI, or you can use `map` from the command line with the `--profile` flag to generate a profile file which can be opened and analysed later (useful for large or long-running jobs):

```
module load arm-forge/22.1.1
map --profile mpirun my_mpi_app
```

You can also isolate a particular section of the code by compiling with the `arm-forge` module loaded, and calling (once each) `allinea_start_sampling()` and `allinea_stop_sampling()` from the header `"mapsampler_api.h"`.

Arm MAP

Try profiling the `list_append` code again using Arm MAP to compare against `gprof`.

Once you're familiar with it, try using it to speed up the matrix-matrix multiplication code in the notebook.

See https://developer.arm.com/documentation/101136/22-1-1/MAP/Get-started-with-MAP/Welcome-page?lang=en for documentation on Arm MAP.

Summary

Profiling tools can help pinpoint optimisation targets and give information about bottlenecks in a program.

Arm MAP is particularly powerful, especially for parallel applications.

Experience, and an understanding of compilers and computer architecture can both help in figuring out how to speed up an section of code or eliminate a bottleneck.

Sometimes there won't be a particular hotspot, in which case it pays to consider the overall program architecture.

One strategy we haven't discussed much yet is parallelisation.
This can be particularly powerful – stay tuned for the next session!