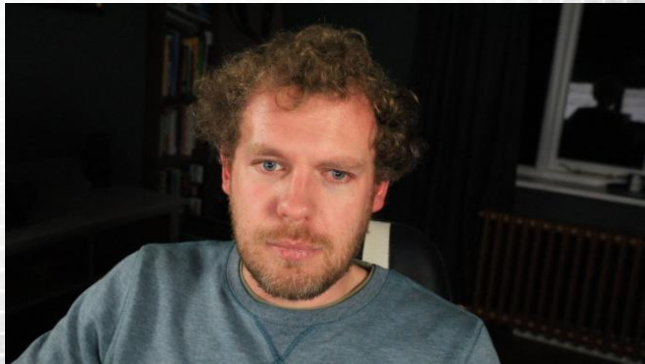
Parallel Computing with GPUs

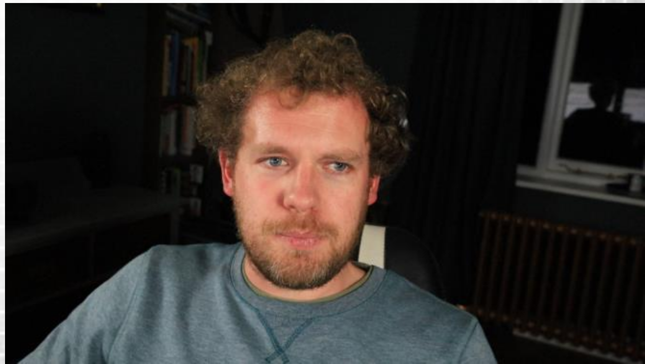
Optimisation
Part 2 - Compute Bound Code

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/



This Lecture (learning objectives)

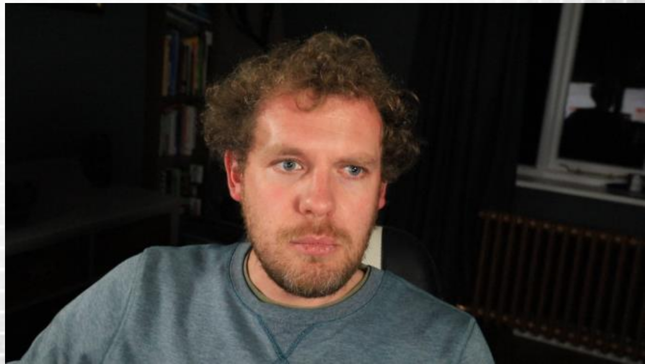
- ❑Compute Bound Code
  - ❑Apply a series of approaches to improve compute bound code performance



Compute Bound: Optimisation

- ❑Approach 1: Compile with full optimisation
  - ❑msvc compiler is very good at optimising code for efficiency
  - ❑Many of the techniques we will examine can be applied automatically by a compiler.
  - ❑Optimisation: Compiler /O Optimisation property
  - ❑Help the compiler
    - ❑Refactor code to make it clear (clear to developers is clear to a compiler)
    - ❑Avoid complicated control flow

Optimisation Level	Description
/O1	Optimises code for minimum size
/O2	Optimises code for maximum speed
/Od	Disables optimisation for debugging
/Oi	Generates intrinsic functions for appropriate calls
/Og	Enables global optimisations

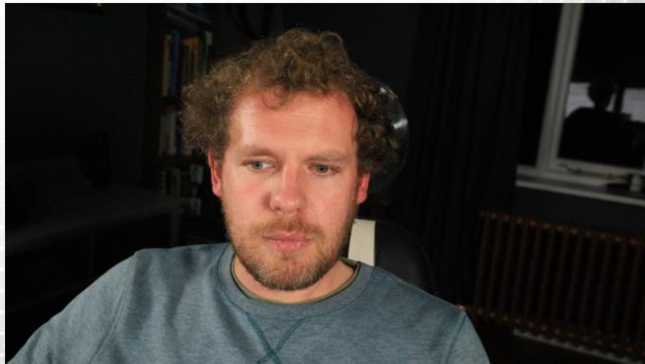


Compute Bound: Optimisation

- Approach 2: Redesign the program
  - Compilers cant do this and it is most likely to have the biggest impact
  - If you have a loop that is executed 1000's of times then find a way to do it without the loop.
  - Be familiar with algorithms
    - Understand big O(n) notation
    - E.g. Sequential search has many faster replacements

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

http://bigocheatsheet.com/

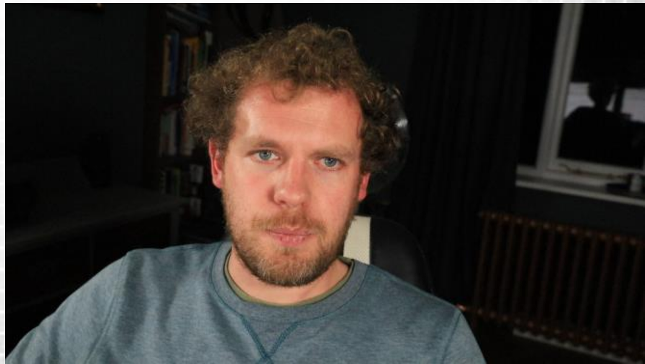


Compute Bound: Optimisations

- Approach 3: Understand operation performance
  - Cost of going to disk is massive
  - Loop Invariant Computations: move operations out of loops where possible
  - Strength reduction: replace expression with cheaper ones

Core i7 Instruction	Cycle Latency
Integer ADD SUB (x32 and x64)	1
Integer MUL (x32 and x64)	3
Integer DIV (x32)	17-28
Integer DIV (x64)	28-90
Floating Point ADD SUB (x32)	3
Floating Point MUL (x32)	5
Floating Point DIV (x32)	7-27

http://www.agner.org/optimize/instruction_tables.pdf



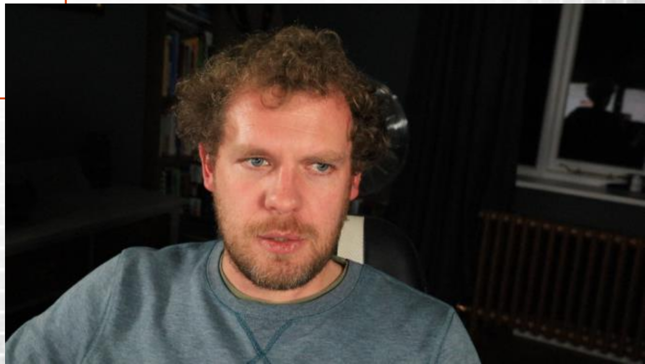
Compute Bound: Optimisations

- Approach 4: function in-lining
  - In-lining increases code size but reduces function calls.
    - Make your simple function a macro
    - Or use the _inline operator
  - Be sensible: Not everything should be in-lined

```
float vec2f_len(vec2f a, vec2f b)
{
    vec2f r;
    r.x = a.x - b.x;
    r.y = a.y b.y;
    return (float)sqrt(r.x*r.x + r.y*r.y);  //requires #include <math.h>
}
```

```
#define vec2f_len(a, b) ((float)sqrt((a.x-b.x)*(a.x-b.x) - (a.y-
b.y)*(a.y-b.y)))
```

```
_inline float vec2f_len(vec2f a, vec2f b)
{
    return (float)sqrt((a.x-b.x)*(a.x-b.x) - (a.y-b.y)*(a.y-b.y));
}
```

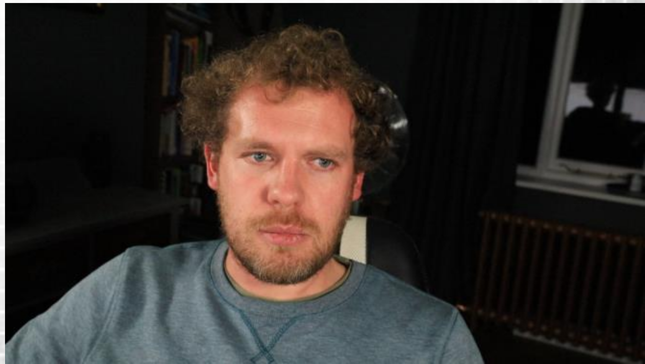


Compute Bound: Optimisations

- Approach 5: Loop unrolling
  - msvc can do this automatically
  - Reduces the number of branch executions

```
for (int i=0; i<100; i++){
    some_function(i);
}
```

```
for (int i=0; i<100;){
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
    some_function(i); i++;
}
```

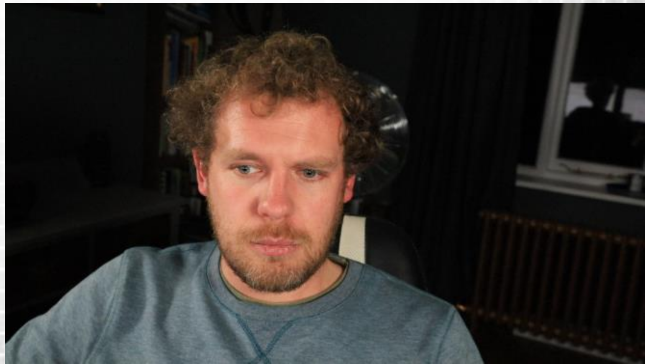


Compute Bound: Optimisations

- Approach 6: Loop jamming
  - Combine adjacent loops to minimise branching (for ranges over the same variable)
  - E.g. Reduction of iterating and testing value `i`

```
for (i=0; i<dim, i++){
    for (j=0;j<dim; j++){
        matrix[i][j] = rand();
    }
}
for (i=0; i<dim, i++){
    matrix[i][i] = 0;
}
```

```
for (i=0; i<dim, i++){
    for (j=0;j<dim; j++){
        matrix[i][j] = rand();
    }
    matrix[i][i] = 0;
}
```

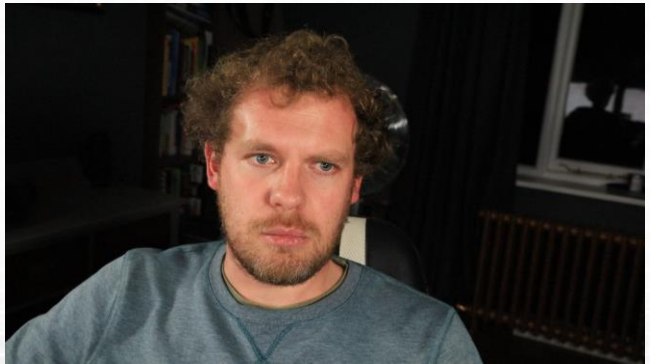


Compute Bound: Optimisations

- Approach 6: Global or heap variables
  - Avoid referencing global or heap variables from within loops
    - Global variables can not be cached in registers
    - Better to write to a local variable
  - Make a local copy of the variable which can be cached
    - Be careful that nothing else requires the variable before you modify it

```
int count;

void test1(void)
{
    int i;
    for(i=0;i<N;i++){
        count += f();
    }
}
void test2(void)
{
    int i, local_count;
    local_count = count;
    for(i=0;i<N;i++){
        local_count += f();
    }
    count = local_count;
}
```



Compute Bound: Optimisations

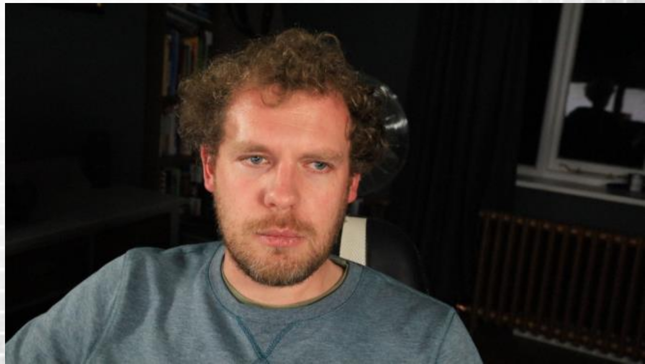
- Approach 7: Function calls
  - Functions are a good way of modularising code
  - Function calls do however have an overhead
    - Stack and program counter must be manipulated
  - It can be beneficial to avoid function calls within loops

```c
void f()
{
    //lots of work
}

void test_f()
{
    int i;
    for(i=0;i<N;i++){
        f();
    }
}
```

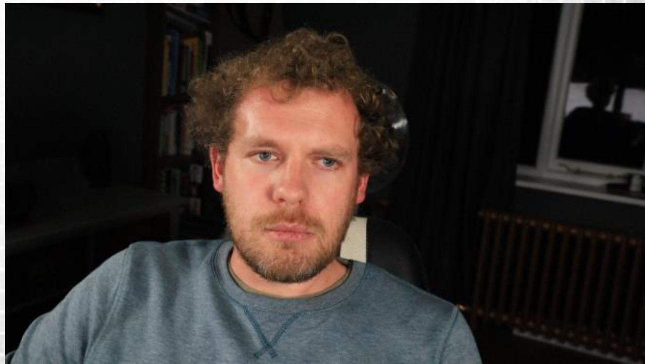
```c
void g()
{
    int i;
    for(i=0;i<N;i++){
        //lots of work
    }
}

void test_g()
{
    g();
}
```



Compute Bound: Optimisations

- Approach 8: Don’t over use the stack
  - Loops rather than recursion
    - C compilers are very good at optimising loops
    - Only certain recursive functions can be optimised
    - Function calls increase stack usage
  - Avoid compile time allocation large structures or arrays on the stack
    - E.g. `int x[10000000];`
    - Use the **heap** or global arrays
  - Avoid passing large structures as arguments
    - They are copied by value
    - Pass a pointer instead



This Lecture (learning objectives)

- ❑Compute Bound Code
  - ❑Apply a series of approaches to improve compute bound code performance