Getting the clowns out of the Mandelbrot

Philipp Czerner

May 7, 2018

Overview

optimisation	time	change
baseline	1	
parallelise	0.34752	+188%
dynamic distribution	0.27227	+28%
enable optimisations	0.21961	+24%
remove complex numbers	0.07595	+189%
remove mod and branch	0.07611	+0%
loop unrolling	0.02933	+159%
main cardioid fastpath	0.00258	+1033%
fractal bulb fastpath	0.00188	+37%
distributed initialisation	0.00177	+6%
total	0.00177	$\times$564.97

Baseline

```c
for (int i = 0; i < size_y; i++) {
  for (uint64_t j = 0; j < size_x; ++j) {
    double x = view_x0 + j * step_x;
    double y = view_y1 - (i + offset_y) * step_y;
    complex double C = x + y * I, Z = C;
    int k;
    for (k = 1; cabs(Z) < 2 && k < max_iter; ++k) {
      Z = Z * Z + C;
    }
    if (k == max_iter) {
      memcpy(image, "\0\0", 3);
    } else {
      int index = (k + palette_shift) % number_of_colors;
      memcpy(image, colors[index], 3);
    }
    image += 3;
  }
}
```

- Some tiny changes to code style and control flow, nothing performance related

Parallelise (+188%)

```
for (int i = 0; i < work_size; ++i) {
  work_args[i] = (Mandelbrot_call_args) {(u8*)image, size_x, size_y, ... };
  work_args[i].offset_y = size_y*i / work_size;
  work_args[i].size_y = size_y*(i+1) / work_size - work_args[i].offset_y;
  work_args[i].image += work_args[i].size_x * work_args[i].offset_y * 3;
}
pthread_t threads[num_threads];
for (int i = 0; i < num_threads; ++i) {
  pthread_create(&threads[i], NULL, &mandelbrot_thread, NULL);
}
for (int i = 0; i < num_threads; ++i) {
  pthread_join(threads[i], NULL);
}
```

- Note that work is spread over threads if uneven
- Always split into contiguous chunks!
  - Here, x-axis is contiguous in memory

Dynamic distribution (+28%)

```
#define work_size 512
Mandelbrot_call_args work_args[work_size];
atomic_int work_index;

void* mandelbrot_thread(void* _) {
  while (work_index < work_size) {
    int i = work_index++;
    if (i >= work_size) break;
    Mandelbrot_call_args* p = &work_args[i];
    mandelbrot_calculate( ... );
  }
}
```

- Synchronisation is just a single integer
- Not much to see here

Enable optimisations (+24%)

```
__attribute__((optimize(3), __target__("avx,ssse3,sse4a")))
void mandelbrot_calculate( ... ) { ... }
```

- Enable via an attribute as I cannot control the compilation process

Remove complex numbers (+189%)

```
for (int j = 0; j < size_x; ++j) {
  double Cr = view_x0 + j * step_x;
  double Ci = y, Zr = 0, Zi = 0, Zr_old = 0;
  int k;
  for (k = 0; Zr*Zr + Zi*Zi < 4 && k < max_iter; ++k) {
    Zr_old = Zr;
    Zr = Zr*Zr - Zi*Zi + Cr;
    Zi = 2.0 * Zr_old * Zi + Ci;
  }
  if (k == max_iter) {
    memcpy(image, "\0\0", 3);
  } else {
    int index = (k + palette_shift) % number_of_colors);
    memcpy(image, colors[index], 3);
  }
  image += 3;
}
```

- Remove the square root inside cabs
- Write down the complex math by hand, simplify squaring a bit

Remove mod and branch (+0%)

```
for (int j = 0; j < size_x; ++j) {
  double Cr = view_x0 + j * step_x;
  double Ci = y, Zr = 0, Zi = 0, Zr_old = 0;
  int k;
  for (k = 0; Zr*Zr + Zi*Zi < 4 && k < max_iter; ++k) {
    Zr_old = Zr;
    Zr = Zr*Zr - Zi*Zi + Cr;
    Zi = 2.0 * Zr_old * Zi + Ci;
  }
  memcpy(image, &colors_opt[3*k], 3);
  image += 3;
}
```

- **Context**: Similar size of `max_iter` and `number_of_colors`
- No performance gain right now, but simplifies code
- Has to be set up by the main thread beforehand (almost no overhead)

Loop unrolling (+159%)

```
...
for (int kk = 0; kk < max_iter; ++kk) {
  u8 flag1 = Z1r*Z1r + Z1i*Z1i < 4;
  u8 flag2 = Z2r*Z2r + Z2i*Z2i < 4;
  u8 flag3 = Z3r*Z3r + Z3i*Z3i < 4;
  u8 flag4 = Z4r*Z4r + Z4i*Z4i < 4;
  if (!(flag1 || flag2 || flag3 || flag4)) break;
  k1 += flag1;   k2 += flag2;   k3 += flag3;   k4 += flag4;
  Z1r_old = Z1r; Z2r_old = Z2r; Z3r_old = Z3r; Z4r_old = Z4r;
  Z1r = Z1r*Z1r - Z1i*Z1i + C1r;
  Z2r = Z2r*Z2r - Z2i*Z2i + C2r;
  Z3r = Z3r*Z3r - Z3i*Z3i + C3r;
  Z4r = Z4r*Z4r - Z4i*Z4i + C4r;
  Z1i = 2.0 * Z1r_old * Z1i + y;
  Z2i = 2.0 * Z2r_old * Z2i + y;
  Z3i = 2.0 * Z3r_old * Z3i + y;
  Z4i = 2.0 * Z4r_old * Z4i + y;
}
...
```

- Manually unroll the loop to process four elements at once.

Loop unrolling ($+159\%$)

- No branches for the finished elements!
  - not trivial to make work, compiler cannot do this
  - if $|Z| > 2$ in some iteration, it will continue to be
- `size_x` may not be a multiple of 4
  - must process remaining elements afterwards

Manual vectorisation

```
typedef double   v4d __attribute__((vector_size (sizeof(double)  *4)));
typedef long int v4i __attribute__((vector_size (sizeof(long int)*4)));
v4d Cr;
for (int m = 0; m < 4; ++m)
  Cr[m] = view_x0 + (j+m) * step_x;
double Ci = y;
v4d Zr = {0}, Zi = {0}, Zr_old = {0};
v4i k = {0};
for (int kk = 0; kk < max_iter; ++kk) {
  v4i tmp = (Zr*Zr + Zi*Zi) < 4;
  k -= tmp; // Note that vector comparison set all bits
  if (!(tmp[0] || tmp[1] || tmp[2] || tmp[3])) break;
  Zr_old = Zr;
  Zr = Zr*Zr - Zi*Zi + Cr;
  Zi = 2.0 * Zr_old * Zi + y;
}
for (int m = 0; m < 4; ++m)
  memcpy(image+3*m, &colors_opt[3*k[m]], 3);
```

- Nice and easy to use

Manual vectorisation

```
typedef double   v4d __attribute__((vector_size (sizeof(double)  *4)));
typedef long int v4i __attribute__((vector_size (sizeof(long int)*4)));
v4d Cr;
for (int m = 0; m < 4; ++m)
  Cr[m] = view_x0 + (j+m) * step_x;
double Ci = y;
v4d Zr = {0}, Zi = {0}, Zr_old = {0};
v4i k = {0};
for (int kk = 0; kk < max_iter; ++kk) {
  v4i tmp = (Zr*Zr + Zi*Zi) < 4;
  k -= tmp; // Note that vector comparison set all bits
  if (!(tmp[0] || tmp[1] || tmp[2] || tmp[3])) break;
  Zr_old = Zr;
  Zr = Zr*Zr - Zi*Zi + Cr;
  Zi = 2.0 * Zr_old * Zi + y;
}
for (int m = 0; m < 4; ++m)
  memcpy(image+3*m, &colors_opt[3*k[m]], 3);
```

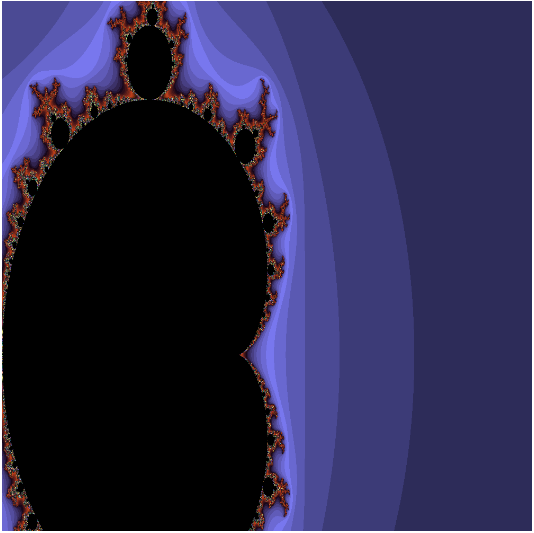
- Nice and easy to use
- Makes things about 12% slower!

Domain-specific optimisations

- Done with routine tricks, now the real work starts
- Bottleneck is in inner loop, CPU-bound
  - Use profiler / custom timing code (see below) to verify
- Try to figure out behaviour of the program, have to look at what the code is actually trying to accomplish

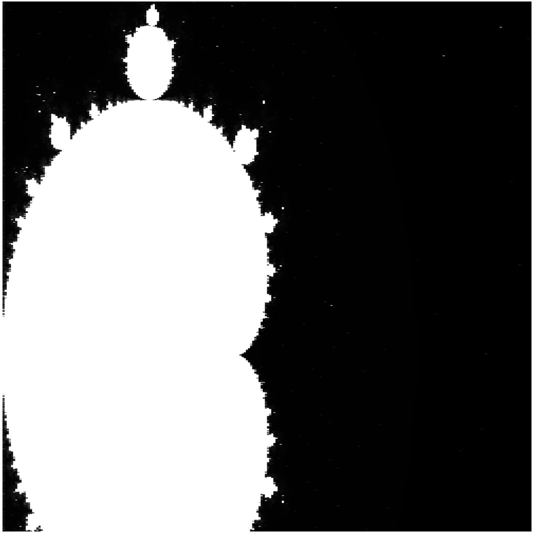
```
// Queries high-precision timer. Not necessarily related to actual
// wall-clock time, use OS facilities for that.
uint64_t rdtsc() {
  uint32_t l, h;
  asm volatile ("rdtscp\nmov %%edx, %0\nmov %%eax, %1\n\t":
                "=r" (h), "=r" (l) :: "%eax", "%edx");
  return (uint64_t)h << 32 | (uint64_t)l;
}
```

Domain-specific optimisations



- Where is the time spent?
  - No need to guess, just measure.

Domain-specific optimisations



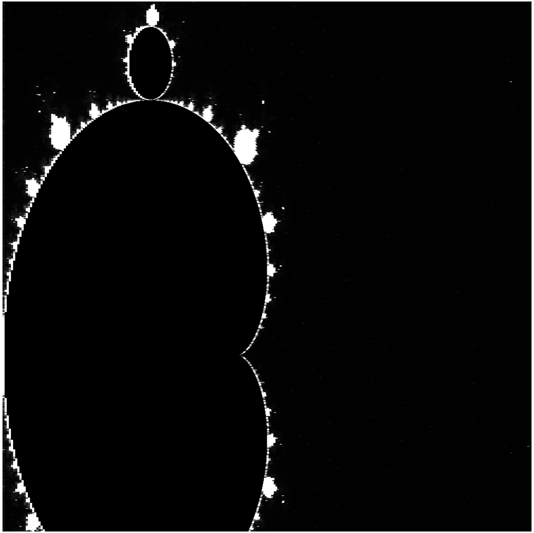
- White is more time
- The main cardioid is about a third, and quite expensive

Main cardioid fastpath (+1033%)

```
u64 j;
for (j = 0; j+4 <= size_x; j += 4) {
  while (j < size_x) {
    double a = view_x0 + j*step_x - 0.25;
    double q = a*a + y*y;
    if (q*(q+a) >= y*y * 0.25) break;
    memcpy(image, "\0\0", 3);
    ++j;
    image += 3;
  }
  if (j+4 > size_x) break;
  ...
}
```

- **Context:** This optimisation is only good when the main cardioid makes up a large part of the image!
- Use formula to detect the inside (I simply looked it up)
- Check is quite cheap, can do it on every pixel
- Same thing for the bulb on the top (additional +37%)

Main cardioid fastpath (+1033%)



- Much better.

Distributed initialisation (+6%)

```
int thread_count;
pthread_t threads[64];
int work_size;
Mandelbrot_call_args work_args[1000];
atomic_int thread_index, work_index;

void* mandelbrot_thread(void* _) {
  while (thread_index < thread_count) {
    int i = thread_index++;
    if (i >= thread_count) break;
    pthread_create(&threads[i], NULL, &mandelbrot_thread, NULL);
  }
  while (work_index < work_size) {
    int i = work_index++;
    if (i >= work_size) break;
    Mandelbrot_call_args* p = &work_args[i];
    mandelbrot_calculate( ... );
  }
}
```

- Let the threads initialise each other

Final notes

- Don't blindly trust the compiler to do non-trivial optimisations—you know much better which transformations are safe
- Look at the actual data, effort spent visualising is seldom wasted
- Code is now fast enough ($\sim$10ms) that constant overheads (initialisation, synchronisation) become relevant again
- No changes to the calculation (e.g. low precision approximation or such) as Mandelbrot is sensitive to rounding errors