Code Tuning: the basics

MJ Rutter
mjr19@cam

Lent 2017

Introduction

This talk is an overview of some of the thoughts one might have when considering code tuning. It is based on a case study of a piece of code which was not written in TCM (so I can be rude about it).

The code in question combines C++ and MPI, so one can consider both efficiency of the use of the CPU, and the efficiency of the interprocess communications. This talk does the first half, and ignores the MPI part for now.

The Start

The code in question compiles without error, and is designed to run even with just a single process. When using OpenMPI it is possible to run single process MPI jobs without invoking `mpirun` or `mpiexec`, but just by invoking the binary directly.

The code has no input files, its parameters being compiled in, though it does write some output files. Suitable parameters were provided so that it ran in around 30s.

A Run

```
$ wc -l *.cpp *.h
 1134 Layers3DhexNewGrid.cpp
  172 vema.h
 1306 total
$ mpicpc Layers3DhexNewGrid.cpp
$ mpirun -np 2 ./a.out
[...]
mpirun has exited due to process rank 0 with PID 1301 on
node pc23 exiting improperly. There are three reasons this could occur:
[...]
2. this process called "init", but exited without calling "finalize".
By rule, all processes that call "init" MUST call "finalize" prior to
exiting or it will be considered an "abnormal termination"
```

Oh dear

```
$ grep -c MPI_Finalize *.cpp *.h
Layers3DhexNewGrid.cpp:0
vema.h:0
```

Before we worry about performance, we should worry about getting this code to comply with the MPI standard. Adding '`MPI_Finalize();`' before the `return 0;` at the end of `main()` resolves this issue. Without this the code will always exit with an error (confusing for scripts), and the final output is unlikely to be written unless rank zero (which does the writing) completes this before any rank exits without calling `MPI_Finalize()`. One might be lucky...

Why not perfect?

Two issues can lead to a lack of optimality. The first is that the wrong algorithm is being used. This potential issue we do not address here.

The second is that the algorithm is not making the best possible use of the CPU. What can cause this?

We need to recall how a CPU works.

A CPU

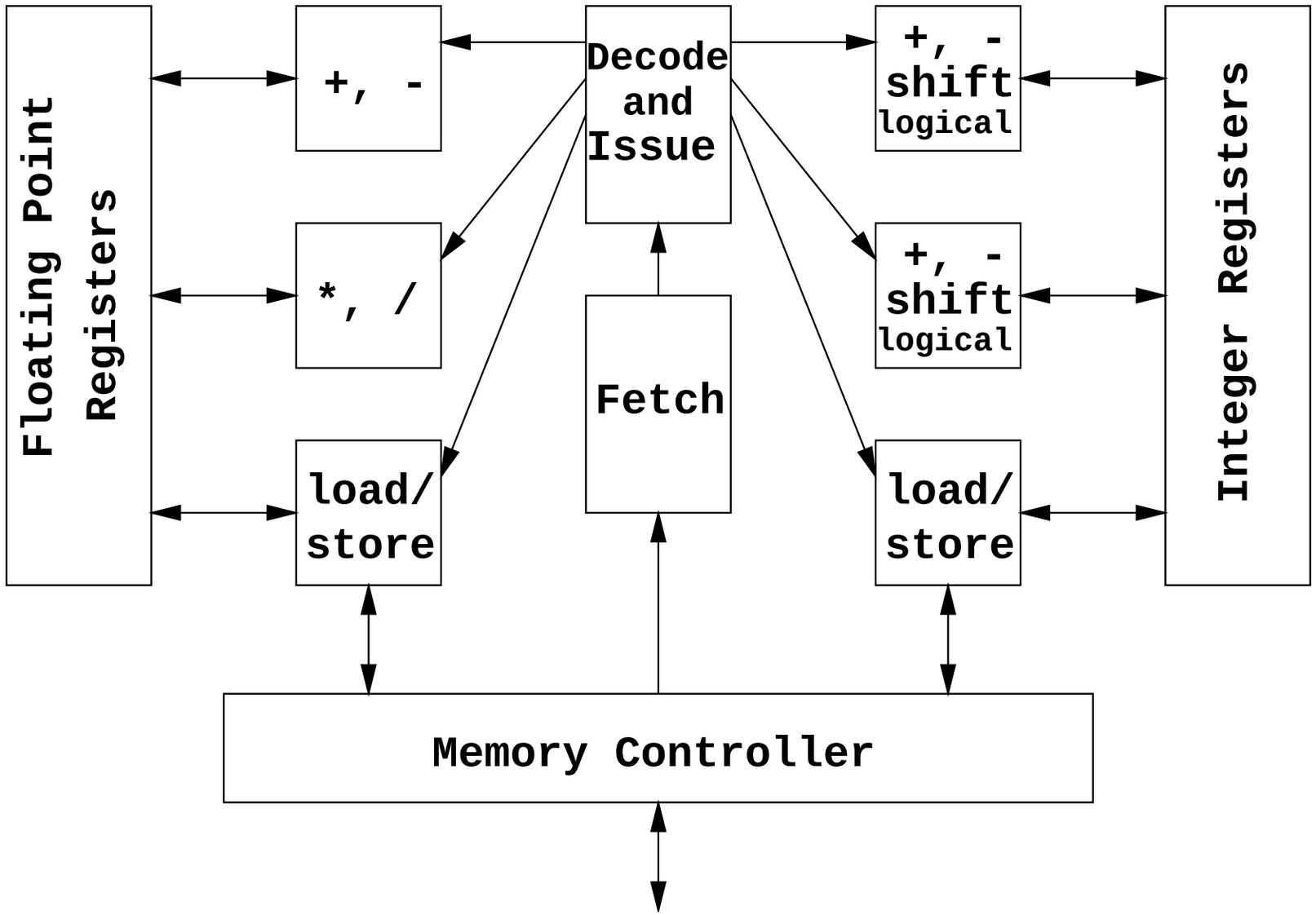
A CPU typically contains multiple functional units. Each is dedicated to one particular task, such as integer addition, floating point multiplication, floating point addition, etc. Ignoring awkward things like division and square roots, in general each functional unit can accept a new instruction every clock cycle, but it may take several clock cycles to produce its result.

The inevitable consequence is that many instructions are in some state of execution, or partial completion, simultaneously, even when one is considering a single core executing serial code.

This can be awkward. To maintain peak FLOPS one might need code which can issue a floating point add and a floating point multiplication on every clock cycle, but which does not need to use the results for another five clock cycles.

If an instruction cannot be started because it depends on the result of a previous instruction which has yet to finish, this is called a data dependency.

Schematic of Typical RISC CPU



Memory

A clock cycle is typically around 0.4ns (2.5GHz). The time it takes memory to respond to an access is typically 25ns. (DDR4/2400 16-16-16 is 26.7ns as 32 cycles of a 1200MHz clock.)

So it can take well over fifty clock cycles for data fetched from main memory to be available to a functional unit.

The maximum bandwidth one could hope for from dual channel DDR4/2400 memory is two channels times 8 bytes per channel times 2400M transfers per second, or 38GB/s. More usefully, this is 4.8G double precision floating point values per second, or two double precision floating point values per clock cycle, assuming a 2.4GHz CPU clock speed.

A floating point operation could require three memory transfers (two items of data in, one out). One must hope that in practice these transfers are from registers or cache, not from main memory, for the main memory bandwidth is typically insufficient to sustain one floating point operation per clock cycle per CPU. As we will see in detail shortly, the best desktop CPUs in TCM are capable of 64 floating point operations per clock cycle across their four cores.

Division

Division is nasty. It takes around two dozen clock-cycles to complete, which is about four times as long as addition or multiplication. But, unlike addition and multiplication, a new divide cannot start on every clock cycle, but only once the previous one has finished.

Worse, generally it has no dedicated functional unit, and instead ties up the floating point addition and multiplication units whilst it operates. (It generally operates by using a Newton-Raphson like method to find the reciprocal of of the divisor, and then doing a final multiplication.)

Nasty, but not as nasty as a memory cache miss. And divisions are easier to spot in code than cache misses!

Idleness

If a process is expected to be idle waiting for something for a long time, the CPU can switch to running a different process, or simply become idle.

However, a process switch typically takes a few microseconds. If the expected wait is of the order of milliseconds (hard disk drive response time) or seconds (human), then a process switch is a good answer. If it is tens of nanoseconds (memory) or less (functional unit latency), then a process switch would not be a good answer. The CPU will remain dedicated to the process, and `top` or `ps` will report this 100% commitment.

Quick Estimates

A quick estimate of whether it is worth trying to optimise serial code can be given by estimating how many GFLOPS it is achieving, and comparing to the maximum theoretical performance of a CPU core. If one is already close to 100%, there is little room for improvement.

In some cases one knows almost exactly how many floating point operations are used by a code, so calculating GFLOPS just requires a stopwatch. Usually one does not, and needs to ask the computer to count them. Modern processors are surprisingly poor at this, but old ones are better, so I have chosen to run this on a Core2:

```
$ mflops ./a.out
[...]
Time:              59.8207s
MFLOPS:             840.872
Vectorisation:         5.8%
Av vector length:      1.03
```

That was run on a 2.4GHz Core2. So how did we do?

Peak FLOPS

The peak number of double precision FLOPS per clock cycle for recent Intel processors is as follows:

Core2	4
Nehalem	4
Sandy / Ivy Bridge	8
Haswell	16

So we managed 0.84GFLOPS from a potential 9.6GFLOPS, or just under 9% of peak. There is the potential to do better.

Better

```
$ mpicpc -Ofast Layers3DhexNewGrid.cpp
$ mflops ./a.out
[...]
Time:              31.1731s
MFLOPS:             1624.64
Vectorisation:         5.7%
Av vector length:      1.03
```

Certainly better – now about 17% of peak.

Compiler Flags

Do not forget compiler flags, but do be careful. This optimisation flag permits algebraic transformations forbidden by the C++ standard, such as division to multiplication by the reciprocal, transformations such as

$$a * b + a * c = a * (b + c)$$

or

$$(a + b) + c = a + (b + c)$$

The transformations are always algebraically valid, but may not be valid for finite precision finite range arithmetic. So they may cause answers to change slightly (or, in rare cases, dramatically).

Vectorisation

When one uses the phrase ‘vector computer’ people think of Crays. The Cray I, introduced in 1975, had vector registers containing 64 floating point numbers each. These then fed functional units capable of starting one operation per cycle, the vector register guaranteeing that 64 independent operations were waiting, with no nasty data dependencies. This idea continued with the Cray X/MP (1982) (MP meaning multiple processors), Y/MP (1992) and J90, and was extended with the Cray C90 (1991) which had 128 element vector registers.

Cray 1 at München

Vectorisation the Intel Way

Intel's approach to vectorisation has been very different. Vectors for double-precision numbers were introduced with the Pentium 4 with its SSE2 instructions. Since the Core2 was introduced in 2006, the functional units have been able to operate on a whole vector at once.

With separate units for addition and multiplication, one full vector of adds and one full vector of multiplications can be started each clock cycle.

The bad news is that the vector length is just two elements for the Core2 and Nehalem.

The better news is that the AVX instructions introduced with the Sandy Bridge increase the vector length to four elements.

AVX-512 (not yet generally available) will increase it again to eight elements (eight times 64 bits = 512).

FMA

Fused Multiply Add.

A common operation (in dot products and matrix multiplications) is

$$x = x + y * z$$

This Intel introduced as a single instruction on its Haswell processors. It also ensured that two of these instructions, each operating on a full four-element vector, could be started each clock cycle. As the instruction itself contains two floating point operations, that is sixteen FP operations per cycle.

Keep Up!

If your code uses just SSE2 instructions, it will never achieve more than four FP operations per clock cycle.

If your code uses AVX instructions, it might achieve eight FP operations per cycle, but it won't run at all on Core2s or Nehalems.

If your code uses AVX2 instructions, it might achieve 16 FP operations per cycle, but it won't run on anything older than Haswells.

Intel's maths kernel library will detect the processor it is running on, and execute different code accordingly. Intel's compiler can also generate code which does this, if one uses options such as

```
-axSSSE3,AVX,CORE-AVX2
```

then three variants will be produced, targetted at Core2 (and Nehalem), Sandy (and Ivy) Bridges, and Haswell.

Vectorisation by the Compiler

The approach that Intel's compiler takes for vectorisation is not based on the idea of a particular datatype, such as a complex number or a small vector, filling a vector register.

Rather it thinks in terms of doing multiple iterations of an inner loop simultaneously, with each element of the vector working on data for a different iteration. So, if one has four element vectors, one does four iterations of an inner loop at once. As long as the iteration count of the inner loop is moderately large, this approach works efficiently with lots of different vector lengths provided by the hardware.

More vectorisation

For a loop to be vectorisable, to a first approximation, the loop should:

- have no conditionals
- have no data dependencies between iterations
- call no functions
- have iterations act on consecutive elements of an array

Some of these restrictions can be relaxed slightly: reduction operations, such as dot products, can be vectorised despite the dependency on the accumulated sum. Non-consecutive array access can be vectorised, but with a performance penalty.

Our Code

Is a 3D finite element code in C++. It defines classes for 3D vectors and matrices.

```
class Vector{
public:
        double x, y, z;
        Vector(): x(0.0), y(0.0), z(0.0) {};
        Vector(double ax, double ay, double az): x(ax), y(ay), z(az) {};
[...]
        Vector& operator+= (const Vector& b){
                x += b.x;
                y += b.y;
                z += b.z;
                return *this;
        }
        Vector operator+ (const Vector& b){
                Vector r = *this;
                return r += b;
        }
[...]
}
```

```
class Matrix{
public:
  double a, b, c,
         d, e, f,
         g, h, i;

  Matrix(): a(1.0), b(0.0), c(0.0),
            d(0.0), e(1.0), f(0.0),
            g(0.0), h(0.0), i(1.0) {};
[...]
  double det(){
    return a*e*i - a*f*h - b*d*i + b*f*g + c*d*h - c*e*g;
  }
  double trace(){
    return a + e + i;
  }
  Matrix prod(const Matrix& n) {
    return Matrix( a*n.a+b*n.d+c*n.g, a*n.b+b*n.e+c*n.h, a*n.c+b*n.f+c*n.i,
                   d*n.a+e*n.d+f*n.g, d*n.b+e*n.e+f*n.h, d*n.c+e*n.f+f*n.i,
                   g*n.a+h*n.d+i*n.g, g*n.b+h*n.e+i*n.h, g*n.c+h*n.f+i*n.i );
  }
 [...]
}
```

Unvectorisable!

Whilst in theory one could produce vectorised code for these classes, it is not how Intel's compiler thinks, so it doesn't. The two main loops in the code are too complex to vectorise. So we need to understand that the previous table should have read:

Processor	Peak FP ops/cycle	
	vectorisable	unvectorisable
Pentium 4	2	2
Core2	4	2
Nehalem	4	2
Sandy / Ivy Bridge	8	2
Haswell	16	2

This is not the end of the world. Other architectural improvements mean that unvectorised code will run faster on a Haswell than a Pentium 4, even at the same clock speed.

Ducking Out

So now we understand that our 1.6 GFLOPS is not 17% of a potential 9.6 GFLOPS but rather about 33% of a potential 4.8 GFLOPS unless we are prepared to tackle the vectorisation issue.

The problem with trying to vectorise a datatype is that one needs to do it completely. If part of a loop has the datatype stored in a vector register, and part of the loop treats it as a bunch of scalars, swapping between the two internal representations is likely to take as much time, or more, than one saves by partial use of the vector form.

Key-hole optimisation

```
Matrix B,I;
double mu,K,J,Ja

[...]

Matrix T = (B - I*B.trace()/3.0)*mu/pow(J,5.0/3.0) + I*K*(Ja - 1.0);
```

The Matrix class overloads the + and − operators for acting on two Matrices. It does not define the concept of adding a scalar to a Matrix, so here we see (twice) a scalar being multiplied by the identity matrix, I, to convert it to something which can be added to a matrix.

Unless the compiler is *very* good at spotting all those zeros, this has taken three additions, and replaced them with nine multiplications followed by nine additions. At the risk of offending mathematicians, we need to define a way of adding a scalar to a matrix.

Doing Less

```
Matrix& operator+= (const double& x){
  a += x;
  e += x;
  i += x;
  return *this;
}
Matrix operator+ (const double& x){
  Matrix r = *this;
  return r += x;
}
```

And similarly for $-$, then we can write

```
Matrix T = (B - B.trace()/3.0)*mu/pow(J,5.0/3.0) + K*(Ja - 1.0);
```

Less Done

```
$ mpicc -Ofast Layers3DhexNewGrid.cpp
$ mflops -v ./a.out

                                FP_COMP_OPS_EXE:      49200469177
           SIMD_COMP_INST_RETIRED.PACKED_DOUBLE:       1447234773

Time:              31.1876s
Total FP ops:  5.06477e+10
MFLOPS:            1623.97
Vectorisation:        5.7%
Av vector length:     1.03
```

After Modification

```
                                FP_COMP_OPS_EXE:      48994479939
          SIMD_COMP_INST_RETIRED.PACKED_DOUBLE:       1447234790

Time:              31.5283s
Total FP ops:  5.04417e+10
MFLOPS:            1599.89
Vectorisation:        5.7%
Av vector length:     1.03
```

Though the number of FP ops reported does vary between 'identical' runs, this reduction of over 2×10^8 operations is statistically significant. However, the change in run-time is somewhere between insignificant and in the wrong direction.

We have failed to identify where the time is really being spent.

More Keys

One should also note that

```
Matrix B;
double mu,K,J,Ja

[...]

Matrix T = (B - B.trace()/3.0)*mu/pow(J,5.0/3.0) + K*(Ja - 1.0);
```

will be interpreted as

```
Matrix T = ((B - B.trace()/3.0)*mu)/pow(J,5.0/3.0) + K*(Ja - 1.0);
```

so is a matrix-double multiplication, followed by a matrix-double division.

We probably meant

```
Matrix T = (B - B.trace()/3.0)*(mu/pow(J,5.0/3.0)) + K*(Ja - 1.0);
```

which is now a double-double division, and a matrix-double division.

The compiler has little way of knowing, considering that we have defined the / and * operators here, that that is so. One might also wonder whether

```
Matrix T = (B - B.trace()/3.0)*(mu*pow(J,-5.0/3.0)) + K*(Ja - 1.0);
```

would not be better.

Adding brackets: 31.1s and 5.01e+10 FP ops.
Division to multiplication: 30.0s and 5.01e+10 FP ops.

Finding time

```
$ mpicc -Ofast -g Layers3DhexNewGrid.cpp
$ operf ./a.out
$ opreport -gdf | op2calltree
[HUGE amount of unnecessary output]
$ kcachegrind oprof.out.unnamed &
```

Note the addition of `-g` to the compile line so that the compiler includes information about the instruction to source line-number mapping.

oprof.out.unnamed

File View Go Settings Help

Open | Back | Forward | Up | % Relative | Cycle Detection | Relative to Parent | <> Shorten Templates | CPU_CLK_UNHALTED

Flat Profile

Search: (No Grouping)

Self	Function	Location
85.38	main	a.out: Layers
12.50	__libm_pow_e7	a.out
1.09	Matrix::trace()	a.out: vema.l
0.29	pow	a.out
0.16	closestPointTriangle(Vector...	a.out: Layers
0.15	/no-vmlinux	no-vmlinux
0.11	createNNLtriangle(std::vec...	a.out: Layers
0.06	__memset_sse2	libc-2.23.so:
0.05	/usr/lib/x86_64-linux-gnu/li...	libstdc++.so
0.03	__memcpy_sse2_unaligned	libc-2.23.so:
0.03	__printf_fp	libc-2.23.so
0.02	/usr/lib/x86_64-linux-gnu/li...	libpciaccess.s
0.02	triNodes(int, int, int&, int&,...	a.out: Layers
0.01	vfprintf	libc-2.23.so:
0.01	memchr	libc-2.23.so:
0.01	/lib/x86_64-linux-gnu/libz.s...	libz.so.1.2.8
0.01	__mpn_mul_1	libc-2.23.so:
0.00	__GI_strchr	libc-2.23.so:
0.00	__GI____strtoull_l_internal	libc-2.23.so
0.00	strlen	ld-2.23.so: st
0.00	_int_malloc	libc-2.23.so:
0.00	do_lookup_x	ld-2.23.so: dl
0.00	PMPI_Reduce	a.out
0.00	PMPI_Allreduce	a.out
0.00	strchrnul	libc-2.23.so:
0.00	uselocale	libc-2.23.so
0.00	_intel_fast_memcpy	a.out
0.00	malloc	libc-2.23.so:
0.00	opal_datatype_copy_conte...	a.out
0.00	_IO_old_init	libc-2.23.so:
0.00	writePov(Vector*, double*, ...	a.out: Layers
0.00	__intel_ssse3_memcpy	a.out
0.00	vsnprintf	libc-2.23.so
0.00	PMPI_Allgather	a.out

oprof.out.unnamed [1] - Total CPU_CLK_UNHALTED Cost: 735 413

main

Types | Callers | All Callers | Callee Map | Source Code

#	CPU_CLK_UNHALTED	Source
517		
518	0.66	Matrix Ar = Matrix(Ut0[n2] - Ut0[n1], Ut0[n3] - Ut0[n1], Ut0[n4] - Ut0[n1]);
519		Ar = G.prod(Ar);
520		
521		Vector x1 = Ut[n2] - Ut[n1];
...		...
528		Vector N4 = (x2 - x3).cross(x1 - x3);
529		
530		Matrix A = Matrix(x1, x2, x3);
531	0.17	double vol = A.det()/6.0;
532		Matrix F = A.prod(Ar.inv());
533	1.00	Matrix B = F.prod(F.trans());
534		double J = F.det();
535		double powJ23 = 1.0 + 2.0/3.0*(J - 1.0) - 1.0/9.0*(J - 1.0)*(J - 1.0);
536		
537		double J1, J2, J3, J4;
538	0.16	if (j < ns) {
539	2.42	J1 = vnt[n1]/vn0t[n1];
540	4.93	J2 = vnt[n2]/vn0t[n2];
541	0.69	J3 = vnt[n3]/vn0t[n3];
542	1.70	J4 = vnt[n4]/vn0t[n4];
543		} else {
544	1.65	J1 = vns[n1]/vn0s[n1];
545	1.88	J2 = vns[n2]/vn0s[n2];
546	1.83	J3 = vns[n3]/vn0s[n3];
547	1.80	J4 = vns[n4]/vn0s[n4];
548		}
549	1.14	double Ja = (J1 + J2 + J3 + J4)/4.0;
550		
551		// Neo-Hookean
552	1.65	Matrix T =(B - B.trace()/3.0)*(mu*pow(J,-5.0/3.0)) + K*(Ja - 1.0);
553		
554		// Mooney-Rivlin (C01 = C10 = mu/4)
555		//double I1b = B.trace()/powJ23;
556		//double I2b = 0.5*(I1b*I1b - (B.prod(B).trace())/(powJ23*powJ23));
557		//Matrix T = (B*(1.0 + I1b)/powJ23 - B.prod(B)/(powJ23*powJ23) - I*(I1b + 2....
558		
559	0.24	if (timeStep%di == 0) {
560		//double Ws = 0.5*mu*(B.trace()/powJ23 - 3.0); // Neo-Hookean

#	CPU_CLK_UNHALTED	Hex		Assembly Instructions		Source Position
40 766A		f2 0f 59 0d c6 74 2b	mulsd	0x2b74c6(%rip),%xmm1	# 6beb3...	
40 7671		00				
40 7672	0.00	f2 0f 10 84 24 30 04	movsd	0x430(%rsp),%xmm0		Layers3DhexNewGrid.cp...
40 7679		00 00				
40 767B		f2 0f 59 05 cd 74 2b	mulsd	0x2b74cd(%rip),%xmm0	# 6beb5...	

Parts | Callees | Call Graph | All Callees | Caller Map | Machine Code

Dodgy Data

The breakdown on the left, 85.4% of time spent in `main`, 12.5% in `__libm_pow_e7`, 1.1% in `Matrix::trace()`, 0.3% in `pow` is probably fairly reliable.

But there are some things to note.

`__libm_pow_e7` is presumably a function internal to `pow`, so we should understand this as 12.8% of time spent in `pow`.

It is slightly odd that all of the functions in the Vector and Matrix classes seem to have been inlined, and so do not occur in this list, except for `Matrix::trace()`, which is actually a rather simple function.

Dodgier Data

The data on the right are more suspect. Exact mappings between instructions (which overlap in time anyway) and line numbers are hard at high optimisation levels. In many cases the mapping is only approximately correct.

It is possible that the identical lines

```
2.42     J1 = vnt[n1]/vn0t[n1];
4.93     J2 = vnt[n2]/vn0t[n2];
0.69     J3 = vnt[n3]/vn0t[n3];
1.70     J4 = vnt[n4]/vn0t[n4];
```

do take the very different execution times suggested due to consistent differences in the caching of the different variables. But I find it a little unlikely.

Reading between the lines

That the line

```
Matrix T =(B - B.trace()/3.0)*(mu*pow(J,-5.0/3.0)) +  K*(Ja - 1.0);
```

is taking a mere 1.65% of the total runtime is almost credible once one understands what that means. The call to `pow` will be attributed to the `pow` function in the maths library, not this line. Similarly the call to the trace function, the matrix-double subtraction, the matrix-double multiplication, and the matrix-double addition will all get attributed to the corresponding lines in the definitions of those functions.

The actual work in this line is evaluating

```
K*(Ja - 1.0)
```

multiplying the result of the trace by a third, multiplying the result of the `pow` by `mu`, and an unknown amount of overspill from neighbouring areas.

Not Useless

Though the data are not perfect, they are also not useless. For code which does not have large numbers of functions at critical points, the estimates can be very useful.

There are gaps. The `pow` function is called at four points in the code, and the total time spent there is about 12%. Is that 3% from each call, or 12% from one, and almost nothing from the other three? We do not know.

There are other analysis tools which might help to answer some of these questions, but that is another talk...

The report in kcachegrind seems to suggest that no time was taken on the lines

```
double J = F.det();
double powJ23 = 1.0 + 2.0/3.0*(J - 1.0) - 1.0/9.0*(J - 1.0)*(J - 1.0);
```

Do we believe that? Yes! `powJ23` is not used further, and `J` is used only in its calculation, so the optimising compiler has noticed and eliminated that unnecessary code.

Tension

If one compiles with optimisation turned firmly off, `-O0`, the correspondence between instructions and line numbers is much easier to understand, and the output of tools such as kcachegrind similarly easier to understand.

But one is not interested in optimising one's code subject to it being compiled with `-O0`. One is interested in optimising one's code subject to it being compiled with full optimisation.

A traditional approach to languages such as Fortran results in code where the degree of transformation that the compiler does on full optisation is relatively minor compared with C++, especially C++ with templates. For the C++ code considered here, run-time with full optimisation is around 30s, with default optimisation around 60s, and with no optimisation around 360s. So the difference is a factor of six between no optimisation and full optimisation, and it is quite possible that the most expensive part of the code with no optimisation becomes irrelevant with full optimisation, and vice versa.

If one considers some very traditional code, Linpack 2000x2000 written in Fortran, the difference between no and full optimisation on the same machine with the same compiler suite as used for these C++ examples is a factor of 1.85. The difference between default and full optimisation is not measurable.

Addendum: the Problem with `mflops`

Intel CPUs provide hardware counters which count certain events. These are very low overhead, and very accurate. But one is restricted to the events that Intel has provided.

In general one can count all sorts of cache misses and branch prediction failures, but counting mflops is surprisingly hard. On modern processors one can count floating point instruction issues, which seems like the correct thing to do, but it isn't.

Modern processors issue instructions aggressively in a speculative fashion, and, if data are not actually yet read for the instruction, abandon the instruction and re-issue it. What one wants to count is instructions which complete successfully (known as retirement), not instructions which started, but maybe did nothing.

Addendum: the Data

Running Streams on an Ivy Bridge gives:
Array size 200: 2,244 SSE_FP_PACKED_DOUBLE
Array size 2,000: 24,666 SSE_FP_PACKED_DOUBLE
Array size 20,000: 290,123 SSE_FP_PACKED_DOUBLE
Array size 200,000: 3,725,287 SSE_FP_PACKED_DOUBLE
Array size 2,000,000: 38,782,770 SSE_FP_PACKED_DOUBLE

The superlinear increase in the number of SSE instructions executed with array size would be hard to explain on a retirement basis. Haswell is much worse.

(Expected number of ops: four passes done, of copy, scale, add and triad, and one pre-pass of scale, so $17\times$ array size, or $8\frac{1}{2}$ times array size as packed double ops.)

On a Core2, where the counter is called `SIMD_COMP_INST_RETIRED.PACKED_DOUBLE`, the corresponding results were 2,119, 20,119, 200,119, 2,000,123, 20,000,131. This is beautifully linear, and it shows that we misunderstand how many packed double instructions to expect...

On a Nehalem things are almost linear: array size 20,000, ops 201,345; array size 2,000,000, ops 20,027,795.