Computer Hardware

MJ Rutter
mjr19@cam

Michaelmas 2018

Typeset by FoilTeX

©2018 MJ Rutter
(Version with hyperlinks)

Contents

History 4
The CPU 12
- instructions . . . 18
- pipelines . . . 19
- vector computers . . . 38
- performance measures . . . 39

Memory 44
- DRAM . . . 45
- caches . . . 58

Memory Access Patterns in Practice 86
- matrix multiplication . . . 86
- matrix transposition . . . 110

Vectorisation and GPUs 120
- Vectorisation . . . 122
- Hyperthreading . . . 135
- GPUs . . . 140

Memory Management 152
- virtual addressing . . . 153
- paging to disk . . . 162
- memory segments . . . 171

Compilers & Optimisation 192
- optimisation . . . 193
- the pitfalls of F90 . . . 219

Intel's Evolution 232
Index 274

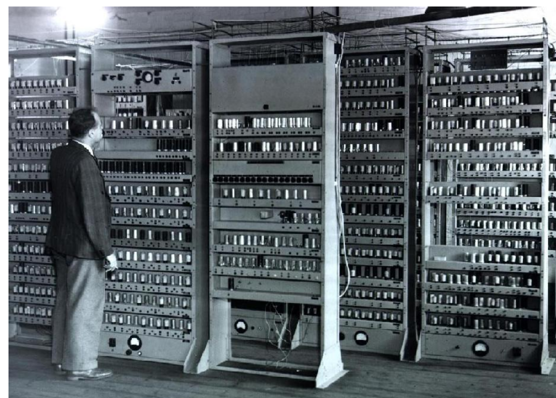
Bibliography 277

History

History in Cambridge: EDSAC

In 1937 the ‘Mathematical Laboratory’ was founded to ‘provide a computing service for general use, and to be a centre for the development of computational techniques in the University.’ The first Director was the chemist Prof Lennard-Jones, now remembered mainly for his interatomic potential, and the computing machines would have been mechanical calculators. Before this laboratory was properly opened, the Second World War intervened.

Towards the end of 1946 the Mathematical Laboratory started to build an Electronic Delay Storage Automatic Calculator, EDSAC. On 6th May 1949 this ran its first program, computing the squares of the integers from 0 to 99. It was programmable in a form of assembler, and already the concepts of subroutines and libraries were being discussed.



Further History in Cambridge

In Cambridge computing in its early years was not regarded as solely a subject to study for its own sake. It was regarded as a tool and service to many other areas of science. By 1953 EDSAC was being used for theoretical chemistry, X-ray molecular biology, numerical analysis, meteorology and radioastronomy. It was able to perform Fourier transforms and Runge-Kutta integration of differential equations. It was the first electronic computer to hold the record for finding the largest known prime number, proving $180 \times (2^{127} - 1)^2 + 1$ prime in 1951.

The theory of computing was not ignored, for a one year post-graduate diploma course in 'Numerical Analysis and Automatic Computing' was launched in 1953, the first formal course in computing leading to a university qualification anywhere in the world. This course, later renamed the Diploma in Computer Science, ran until 2008.

EDSAC led to the creation of LEO (Lyons Electronic Office) which automated payroll, inventory and some aspects of stock control and ordering for the J Lyons & Co. This was the first business computer, and was doing useful work by late 1951.

Meanwhile, Cambridge produced EDSAC 2.

EDSAC 2 to Phoenix

EDSAC 2 moved from mercury tubes for memory to ferrite cores – smaller, faster, and a lot less dangerous (although the poisonous nature of mercury was not fully understood then). It also added support for floating point numbers, by the use of what would now be called microcode instructions. An integer addition could be as fast as 20μs.

EDSAC 2 entered service in 1958, and was the last fully Cambridge-built computer. Its successor, Titan (1964), was a collaboration between Cambridge and Ferranti. It had a rudimentary operating system, and supported high level languages, including, eventually, Fortran.

The 1950s and 1960s Outside of Cambridge

Whereas the 1950s was mostly a decade of prototypes, the 1960s saw the introduction of many things which are clearly ancestors of things common today.

IBM started work on Fortran in the mid 1950s, and Fortran IV appeared in 1961. A year later IBM produced a hard disk drive with flying heads – very similar technology to that in hard drives today.

By the end of the 1960s multitasking, virtual memory, paging, BASIC, ASCII, cache memory, computer algebra, computers with multiple processors (SMP), and even the mouse had appeared.

Phoenix in Cambridge

Titan's successor, Phoenix, was simply a standard IBM, albeit with a Cambridge-written OS (also called Phoenix) on top of IBM's MVS. It arrived in 1971, and remained in use (with substantial upgrades) until 1995. It was the last computer housed by the UCS to be water cooled. Thus sometimes plumbers had to be called to attend to its needs.

See IEEE Annals of the History of Computing, Vol 14 issue 4 (1992) for several articles on the early history of computing in Cambridge.

The 1970s and Beyond Outside of Cambridge

During the 1970s a huge amount more of the technology we recognise today first appeared. UNIX, C, ethernet, TCP/IP, TEX, the first email, the first word-processor, DRAM with one transistor per bit, error correcting memory, and the laser printer.

In terms of radically new ideas, there has been rather little since. The 1980s did add X11 – the remote display of graphical applications between machines from different vendors, and it saw the rise of RISC CPUs. It also saw the first commercial MPP parallel computer.

From our point of view, the important story of the 1990s was of two types of parallelism. Firstly commercial processors capable of starting multiple instructions in a single clock cycle (e.g. IBM Power I, 1990), and secondly the invention of MPI for programming MPP machines.

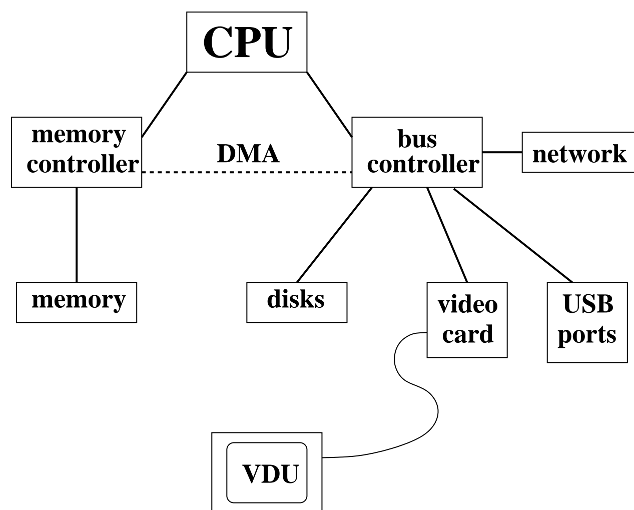
Cambridge's HPCF/S

Phoenix's departure left a large machine room. Into this void, with generous assistance from Hitachi and Research Council funding was born the HPCF (now HPCS). Initially this consisted of a Hitachi S3600, a large (c. 10 tons, c. 60kW) vector machine, and a 96 processor Hitachi SR2201 (a distributed memory machine suited to MPI). The SR2201 would ultimately be upgraded to 256 processors before it was decommissioned.

The HPCF went though various other specialist machines: an SGI Origin 2000, a Power 3 based IBM SP2 and a collection of SunFire 15Ks before moving to clusters of commodity PC servers with high-performance interconnects.

The CPU

Inside the Computer



The Heart of the Computer

The CPU, which for the moment we assume has a single core, is the brains of the computer. Everything else is subordinate to this source of intellect.

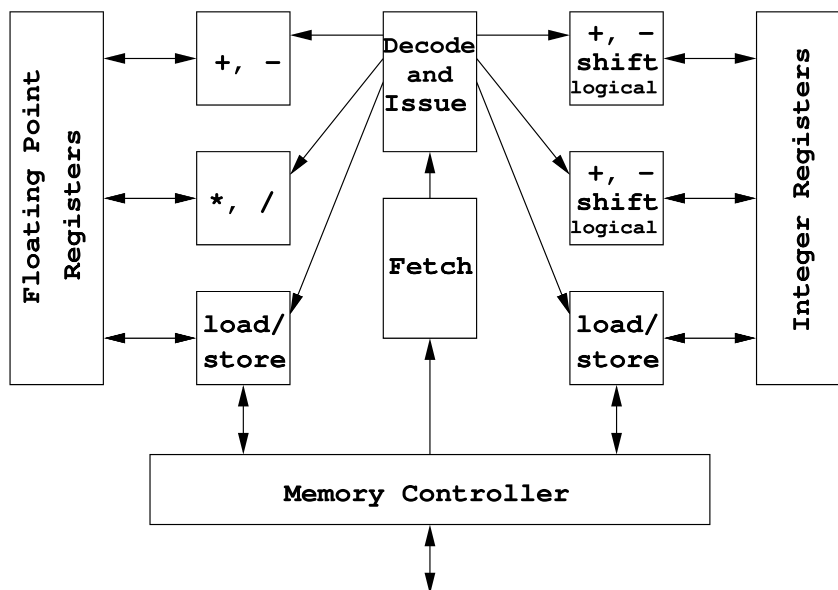
A typical modern CPU understands two main classes of data: integer and floating point. Within those classes it may understand some additional subclasses, such as different precisions.

It can perform basic arithmetic operations and comparisons, governed by a sequence of instructions, or *program*.

It can also perform comparisons, the result of which can change the *execution path* through the program.

Its sole language is machine code, and each family of processors speaks a completely different variant of machine code.

Schematic of Typical RISC CPU



What the bits do

- Memory: not part of the CPU. Used to store both program and data.
- Instruction fetcher: fetches next machine code instruction from memory.
- Instruction decoder: decodes instruction, and sends relevant data on to...
- Functional unit: dedicated to performing a single operation
- Registers: store the input and output of the functional units There are typically about 32 floating point registers, and 32 integer registers.

Partly for historical reasons, there is a separation between the integer and floating point parts of the CPU.

On some CPUs the separation is so strong that the only way of transferring data between the integer and floating point registers is via the memory. On some older CPUs (e.g. the Intel 386), the FPU (floating point unit) is optional and physically distinct. Embedded CPUs, including those in phones and tablets, may lack an FPU.

Basic Operation

'Fetch, Decode, Execute' say all the textbooks.

This is a fairly accurate description of how an individual instruction is processed. To my mind it has an important omission: retire. Retiring an instruction is to acknowledge that it has completed, with its result safety stored in a register or in memory as appropriate.

To keep the different parts of the CPU communicating together, a clock signal is used. This is simply a square wave of a given frequency, and most parts of the CPU will do some useful, observable quantum of work each cycle. The faster the clock signal, the faster a given CPU will operate, within reason. If, for instance, a given CPU expects its integer addition unit to produce a result in a single clock cycle, and that unit takes 0.5ns to produce a result, then clock frequencies of below 2GHz will lead to correct operation, and above 2GHz will produce nonsense.

A square wave has a rising and a falling edge, and both edges may be used internally for synchronisation.

Typical instructions

Integer:

- arithmetic: $+,-,\times,/$, negate
- logical: and, or, not, xor
- bitwise: shift, rotate
- comparison
- load / store (copy between register and memory)

Floating point:

- arithmetic: $+,-,\times,/,\surd$, negate, modulus
- convert to / from integer
- comparison
- load / store (copy between register and memory)

Control:

- (conditional) branch (i.e. goto)

Most modern processors barely distinguish between integers used to represent numbers, and integers used to track memory addresses (i.e. pointers).

A typical instruction

```
fadd f4,f5,f6
```

add the contents of floating point registers 4 and 5, placing the result in register 6, assuming the convention that floating-point instructions are prefixed by ‘f’.

Execution sequence:

- fetch instruction from memory
- decode it
- collect required data (`f4` and `f5`) and send to floating point addition unit
- wait for add to complete
- retrieve result and place in `f6`

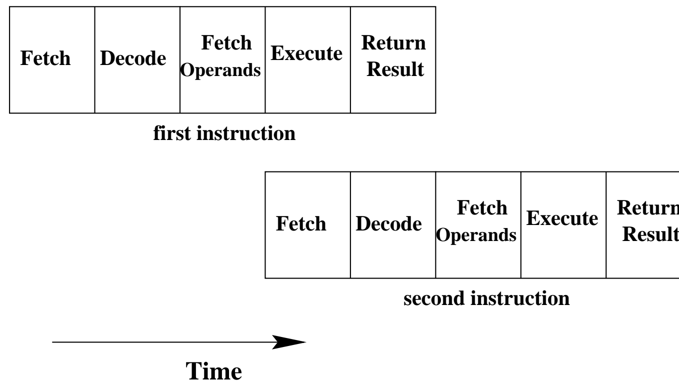
Exact details vary from processor to processor, but always a *pipeline* of operations which must be performed sequentially.

The number of *stages* in the pipeline, or *pipeline depth*, can be between about 5 and 15 depending on the processor.

Making it go faster…

If each pipeline stage takes a single clock-cycle to complete, the previous scheme would suggest that it takes five clock cycles to execute a single instruction.

Clearly one can do better: in the absence of branch instructions, the next instruction can always be both fetched and decoded whilst the previous instruction is executing. This shortens our example to three clock cycles per instruction.



…and faster…

A functional unit may itself be pipelined, if it needs to take more than a single clock-cycle. Considering again floating-point addition, even in base 10 there are three distinct stages to perform:

$9.67 \times 10^5 + 4 \times 10^4$

First the exponents are adjusted so that they are equal:

$9.67 \times 10^5 + 0.4 \times 10^5$

only then can the mantissas be added: 10.07×10^5

then one may have to readjust the exponent: 1.007×10^6

So floating point addition usually takes at least three clock cycles in the execution stage. But the adder may be able to start a new addition ever clock cycle, as these stages use distinct parts of the adder.

Such an adder would have a *latency* of three clock cycles, but a *repeat* or *issue rate* of one clock cycle.

. . . and faster

Further improvements are governed by *data dependency*. Consider:

```
fadd f4,f5,f6
fmul f6,f7,f4
```

(Add `f4` and `f5` placing the result in `f6`, then multiply `f6` and `f7` placing the result back in `f4`.)

Clearly the add must finish (`f6` must be calculated) before the multiply can start. There is a data dependency between the multiply and the add.

But consider

```
fadd f4,f5,f6
fmul f3,f7,f9
```

Now any degree of overlap between these two instructions is permissible: they could even execute simultaneously or in the reverse order and still give the same result.

Breaking Dependencies

```
for(i=0;i<n;i++){          do i=1,n
  sum+=a[i];                 sum=sum+a(i)
}                          enddo
```

This would appear to require three clock cycles per iteration, as the iteration `sum=sum+a[i+1]` cannot start until `sum=sum+a[i]` has completed. However, consider

```
for(i=0;i<n;i+=3){         do i=1,n,3
  s1+=a[i];                  s1=s1+a(i)
  s2+=a[i+1];                s2=s2+a(i+1)
  s3+=a[i+2];                s3=s3+a(i+2)
}                          enddo
sum=s1+s2+s3;              sum=s1+s2+s3
```

The three distinct partial sums have no interdependency, so one add can be issued every cycle.

Do not do this by hand. This is a job for an optimising compiler, as you need to know a lot about the particular processor you are using before you can tell how many paritial sums to use. And worrying about *codas* for `n` not divisible by 3 is tedious.

Register dependencies

```
1. load a[i] into register one
2. multiply register one by register two
3. store register one into a[i]
4. load c[j] into register one
5. add register three to register one
```

As written, no re-ordering of these instructions is possible. The dependencies are not all on data (assuming `a[i]` and `c[j]` are distinct), but the dependency between steps 3 and 4 is on a register which has been re-used.

Most modern processors have more physical registers than they have architectural registers (registers addressed by their instruction set). They map architectural registers onto their physical registers in such a fashion that dependencies such as the above can be broken.

Memory dependencies

```
3. store register one into a[i]
4. load c[j] into register one
```

In C-like languages, it is quite common for the compiler not to know if `a[i]` and `c[j]` refer to the same memory location or not. (In Fortran it is likely that the compiler will know that they are distinct, which is one reason why Fortran is easier for a compiler to optimise.)

Early processors took a conservative approach to memory references, never allowing any read or write to occur before all previous writes had completed. (Re-ordering reads is always safe.)

Many modern processors will allow memory references to be reordered as soon as the addresses are known to be distinct.

Predictions

The pipeline can be quite long in terms of clock cycles. The stages discussed above may each take multiple cycles, and there are stages not discussed above for more complex architectures. For a CPU such as Intel's Haswell the pipeline length is typically around 15 clock cycles. This means that breaking it (refered to as a 'stall') is quite expensive.

One instruction which would be expected to break a pipeline is the conditional jump (or goto) instruction. It means that the next instruction to execute is not next one in memory, which has probably been fetched and decoded, but some other instruction. So modern processors attempt to predict branches, and only stall the pipeline if the prediction is incorrect.

One of the simplest forms of prediction is to assume that backwards conditional jumps are taken, and forward ones not. This works well because the end of a loop is a backwards conditional jump which is generally taken. Another is to assume that the jump will be taken if it was taken last time, and vice versa.

In practice more sophisticated schemes are used which can predict accurately short repeating sequences.

Speculation

A simple CPU does not begin to execute the instruction after the branch until it knows whether the branch was taken: it merely fetches and decodes it, and collects its operands. A further level of sophistication allows the CPU to execute the next instruction(s), provided it is able to throw away all results and side-effects if the branch was mispredicted.

Such execution is called *speculative execution*.

Errors caused by speculated instructions must be carefully discarded. It is no use if

```
if (x>0) x=sqrt(x);
```

causes a crash when the square root is executed speculatively with `x=-1`, nor if

```
if (i<1000) x=a[i];
```

causes a crash when `i=2000` due to trying to access `a[2000]`.

Almost all current processors are capable of some degree of speculation.

OOO!

The final trick to try to keep the pipeline flowing is called out of order execution. If the pipeline is stalled because of a data dependency, then perhaps the next instruction after the stalled instruction can be executed?

Performing such tricks considerably complicates the design of the processor, and anyone thinking in Fortran would ask why the compiler could not have ordered the instructions more appropriately anyway. One answer is that in C-like languages data dependences may not be clear at compile time, as it may only become clear at runtime which pointers are distinct, and which are equivalent.

Superscalar

We have now reached one instruction per cycle, assuming data independency.

If the instructions are short and simple, it is easy for the CPU to dispatch multiple instructions simultaneously, provided that each functional unit receives no more than one instruction per clock cycle.

So, in theory, an FP add, an FP multiply, an integer add, an FP load and an integer store might all be started simultaneously. A processor capable of this is referred to as being *superscalar*. It is scalar (as opposed to being parallel), but it is rather super. All modern processors are superscalar – commercial superscalar CPUs appeared at the start of the 1990s, the Pentium (1993) being Intel's first IA32 example.

RISC instruction sets are carefully designed so that each instruction uses only one functional unit, and it is easy for the decode/issue logic to spot dependencies. CISC is a mess, with a single instruction potentially using several functional units.

CISC (Complex Instruction Set Computer) relies on a single instruction doing a lot of work: maybe incrementing a pointer and loading data from memory and doing an arithmetic operation.

RISC (Reduced Instruction Set Computer) relies on the instructions being very simple – and then letting the CPU overlap them as much as possible.

Superscalar (2)

'All modern processors are superscalar,' so the idea is quite important. Superscalar processors execute serial (i.e. non-parallel) code, but, at the instruction level, they are issuing multiple instructions simultaneously. They have *instruction level parallelism* (ILP).

ILP is measurable for a code: the number of instructions executed divided by the number of clockcycles used. Every time the instruction pipeline stalls, this ratio is in danger of dropping below one. Writing code where it is below 0.1 is not hard. Getting it above half of its theoretical maximum can be quite challenging. A single Intel Haswell core can theoretically start eight instructions per clock cycle, up from six for Sandy Bridge / Ivy Bridge.

Our schematic RISC CPU on page 15 showed two separate integer execution units. This makes sense only for superscalar CPUs, which can then use both at once.

The penalty for pipeline stalls on a superscalar CPU is large in terms of missed opportunities to issue instructions.

An Aside: Choices and Families

There are many choices to make in CPU design. Fixed length instructions, or variable? How many integer registers? How big? How many floating point registers (if any)? Should 'complicated' operations be supported? (Division, square roots, trig. functions, . . .). Should functional units have direct access to memory? Should instructions overwrite an argument with the result? Etc.

This has led to many different CPU families, with no compatibility existing between families, but backwards compatibility within families (newer members can run code compiled for older members).

In the past different families were common in desktop computers. Now the Intel/AMD family has a near monopoly here, but mobile phones usually contain ARM-based CPUs, and printers, routers, cameras etc., often contain MIPS-based CPUs. The Sony PlayStation uses CPUs derived from IBM's Power range, as do the Nintendo Wii and Microsoft Xbox.

At the other end of the computing scale, Intel/AMD has only recently begun to dominate. However, the top twenty machines in the November 2010 Top500 supercomputer list include three using the IBM Power series of processors, and another three using GPUs to assist performance. Back in June 2000, the Top500 list included a single Intel entry, admittedly top, the very specialised one-off ASCI Red. By June 2005 Intel's position had improved to 7 in the top 20.

Compilers

CPUs from different families will speak rather different languages, and, even within a family, new instructions get added from generation to generation to make use of new features.

Hence intelligent Humans write code in well-defined processor-independent languages, such as Fortran or C, and let the compiler do the work of producing the correct instructions for a given CPU. The compiler must also worry quite a lot about interfacing to a given operating system, so running a Windows executable on a machine running MacOS or Linux, even if they have the same CPU, is far from trivial (and generally impossible).

Compilers can, and do, of course, differ in how fast the sequence of instructions they translate code into runs, and even how accurate the translation is.

Well-defined processor-independent languages tend to be supported on by a wide variety of platforms over a long period of time. What I wrote a *long* time ago in Fortran 77 or ANSI C I can still run easily today. What I wrote in QuickBASIC then rewrote in TurboBASIC is now useless again, and became useless remarkably quickly.

Ignoring Intel

Despite Intel's desktop dominance, this course is utterly biased towards discussing RISC machines. It is not fun to explain an instruction such as

```
faddl (%ecx,%eax,8)
```

(add to the register at the top of the FP register stack the value found at the memory address given by the `ecx` register plus $8\times$ the `eax` register) which uses an integer shift ($\times 8$), integer add, FP load and FP add in one instruction.

Furthermore, since the days of the Pentium Pro (1995), Intel's processors have had RISC cores, and a CISC to RISC translator feeding instructions to the core. The RISC core is never exposed to the programmer, leaving Intel free to change it dramatically between processors. A hideous operation like the above will be broken into three or four "μ-ops" for the core. A simpler CISC instruction might map to single μ-op (micro-op).

Designing a CISC core to do a decent degree of pipelining and simultaneous execution, when instructions may use multiple functional units, and memory operations are not neatly separated, is more painful than doing runtime CISC to RISC conversion.

Typical functional unit speeds

Instruction	Latency	Issue rate
iadd/isub	1	1
and, or, etc.	1	1
shift, rotate	1	1
load/store	1-2	1
imul	3-15	3-15
idiv	15-30	6-30
fadd	3	1
fmul	2-3	1
fdiv/fsqrt	12-25	6-25

In general, most things 1 to 3 clock cycles and pipelined, except integer $\times$ and $\div$, and floating point $\div$ and $\surd$.

'Typical' for simple RISC processors. Some processors tend to have longer fp latencies: 4 for `fadd` and `fmul` for the UltraSPARC III, 5 and 7 respectively for the Pentium 4, 3 and 5 respectively for the Core 2 / Nehalem / Sandy Bridge.

Again using the convention that 'f' prefixes floating point instructions, and 'i' integer instructions.

Floating Point Rules?

Those slow integer multiplies are more common that it would seem at first. Consider:

```
double precision x(1000),y(500,500)
```

The address of `x(i)` is the address of `x(1)` plus $8 \times (i-1)$. That multiplication is just a shift. However, `y(i,j)` is that of `y(1,1)` plus $8 \times ((i-1)+(j-1) \times 500)$. A lurking integer multiply!

Compilers may do quite a good job of eliminating unnecessary multiplies from common sequential access patterns.

C does things rather differently, but not necessarily better.

Hard or Soft?

The simple operations, such as $+$, $-$ and $*$ are performed by dedicated pipelined pieces of hardware which typically produce one result each clock cycle, and take around four clock cycles to produce a given result.

Slightly more complicated operations, such as $/$ and $\sqrt{}$ may be done with *microcode*. Microcode is a tiny program on the CPU itself which is executed when a particular instruction, e.g. $/$, is received, and which may use the other hardware units on the CPU multiple times.

Yet more difficult operations, such as trig. functions or logs, are usually done entirely with software in a library. The library uses a collection of power series or rational approximations to the function, and the CPU needs evaluate only the basic arithmetic operations.

The IA32 range is unusual in having microcoded instructions for trig. functions and logs. Even on a Core i7, a single trig instruction can take over 100 clock cycles to execute. RISC CPUs tend to avoid microcode on this scale.

The trig. function instructions date from the old era of the x87 maths coprocessor, and no corresponding instruction exists for data in the newer SSE2/XMM registers.

Division by Multiplication?

There are many ways to perform floating point division. With a fast hardware multiplier, Newton-Raphson like iterative algorithms can be attractive.

$$x_{n+1} = 2x_n - b{x_n}^2$$

will, for reasonable starting guesses, converge to $1/b$. E.g., with $b = 6$.

n	x_n
0	0.2
1	0.16
2	0.1664
3	0.16666624
4	0.1666666666655744

How does one form an initial guess? Remember that the number is already stored as $m \times 2^e$, and $0.5 \leq m < 1$. So a guess of $0.75 \times 2^{1-e}$ is within a factor of 1.5. In practice the first few bits of m are used to index a lookup table to provide the initial guess of the mantissa.

A similar scheme enables one to find $1/\sqrt{b}$, and then $\sqrt{b} = b \times 1/\sqrt{b}$, using the recurrance $x \to 0.5x(3 - bx^2)$

Vector Computers

The phrase ‘vector computer’ means different things to different people.

To Cray, it meant having special ‘vector’ registers which store multiple floating point numbers, ‘multiple’ generally being 64, and on some models 128. These registers could be operated on using single instructions, which would perform element-wise operations on up to the whole register. Usually there would be just a single addition unit, but a vadd instruction would take around 70 clock cycles for a full 64 element register – one cycle per element, and a small pipeline start overhead.

So the idea was that the vector registers gave a simple mechanism for presenting a long sequence of independent operations to a highly pipelined functional unit. Indeed, vector instructions could be ‘chained’, with data dependencies considered on a per element basis, not a per vector basis. So a vadd could start as soon as the first elements of both its operands were available.

To Intel it means having special vector registers which typically hold between two and eight double precision values. Then, as transistors are plentiful, the functional units are designed to act on a whole vector at once, operating on each element in parallel. Indeed, scalar operations proceed by placing just a single element in the vector registers. Although the scalar instructions prevent computations occuring on the unused elements (thus avoiding errors such as divisions by zero occuring in them), they are no faster than the vector instructions which operate on all the elements of the register.

Meaningless Indicators of Performance

The only relevant performance indicator is how long a computer takes to run *your* code. Thus my fastest computer is not necessarily your fastest computer.

Often one buys a computer before one writes the code it has been bought for, so other ‘real-world’ metrics are useful. Some are not useful:

- MHz: the silliest: some CPUs take 4 clock cycles to perform one operation, others perform four operations in one clock cycle. Only any use when comparing otherwise identical CPUs. Even then, it excludes differences in memory performance.
- MIPS: Millions of Instructions Per Second. Theoretical peak speed of decode/issue logic, or maybe the time taken to run a 1970’s benchmark. Gave rise to the name Meaningless Indicator of Performance.
- FLOPS: Floating Point Operations Per Second. Theoretical peak issue rate for floating point computational instructions, ignoring loads and stores and with optimal ratio of $+$ to $*$. Hence MFLOPS, GFLOPS, TFLOPS: 10^6, 10^9, 10^{12} FLOPS.

The Guilty Candidates: Linpack

Linpack 100x100

Solve 100x100 set of double precision linear equations using fixed FORTRAN source. Pity it takes just 0.7 s at 1 MFLOPS and uses under 100KB of memory. Only relevant for pocket calculators.

Linpack 1000x1000 or nxn

Solve 1000x1000 (or nxn) set of double precision linear equations by any means. Usually coded using a blocking method, often in assembler. Is that relevant to your style of coding? Achieving less than 50% of a processor's theoretical peak performance is unusual.

Linpack is convenient in that it has an equal number of adds and multiplies uniformly distributed throughout the code. Thus a CPU with an equal number of FP adders and multipliers, and the ability to issue instructions to all simultaneously, can keep all busy.

Number of operations: O(n^3), memory usage O(n^2).
n chosen by manufacturer to maximise performance, which is reported in GFLOPS.

SPEC

SPEC is a non-profit benchmarking organisation. It has two CPU benchmarking suites, one concentrating on integer performance, and one on floating point. Each consists of around ten programs, and the mean performance is reported.

Unfortunately, the benchmark suites need constant revision to keep ahead of CPU developments. The first was released in 1989, the second in 1992, the third in 1995. None of these use more than 8MB of data, so fit in cache with many current computers. Hence a fourth suite was released in 2000, and then another in 2006, and another in 2017.

It is not possible to compare results from one suite with those from another, and the source is not freely available ($250 for a not–for–profit licence).

SPEC also has a set of throughput benchmarks, SPECrate, which consist of running multiple copies of the serial benchmarks simultaneously. For multicore machines, this shows the contention as the cores compete for limited memory bandwidth. The rules for the 2017 suite allow OpenMP parallelism in the 'serial' scores.

Until 2000, the floating point suite was entirely Fortran.

Two scores are reported, 'base', which permits two optimisation flags to the compiler, and 'peak' which allows any number of compiler flags. Changing the code is not permitted.

SPEC: Standard Performance Evaluation Corporation (`www.spec.org`)

Your Benchmark or Mine?

A couple of years ago, picking the oldest desktop in TCM, a 2.67GHz Pentium 4, and the newest, a 3.1GHz quad core 'Haswell' CPU, I ran two benchmarks.

Linpack gave results of 3.88 GFLOPS for the P4, and 135 GFLOPS for the Haswell, a win for the Haswell by a factor of around 35.

A nasty synthetic integer benchmark I wrote gave run-times of 6.0s on the P4, and 9.7s on the Haswell, a win for the P4 by a factor of 1.6 in speed.

(Linux's notoriously meaningless 'BogoMIPS' benchmark is slightly kinder to the Haswell, giving it a score of 6,185 against 5,350 for the P4.)

It is all too easy for a vendor to use the benchmark of his choice to prove that his computer is faster than a given competitor.

The P4 was a 'Northwood' P4 first sold in 2002, the Haswell was first sold in 2013.

Memory

- DRAM
- Parity and ECC
- Going faster: wide bursts
- Going faster: caches

Memory Design

The first DRAM cell requiring just one transistor and one capacitor to store one bit was invented and produced by Intel in 1974. It was mostly responsible for the early importance of Intel as a chip manufacturer.

The design of DRAM has changed little. The speed, as we shall soon see, has changed little. The price has changed enormously. I can remember when memory cost around £1 per KB (early 1980s). It now costs under 1p per MB, a change of a factor of 10^5, or a little more in real terms. This change in price has allowed a dramatic change in the amount of memory which a computer typically has.

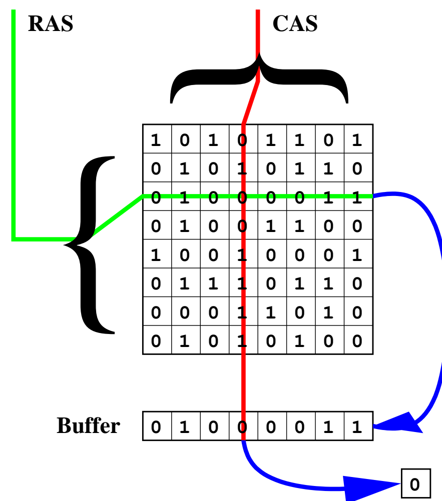
Alternatives to DRAM are SRAM – very fast, but needs six transistors per bit, and flash RAM – unique in retaining data in the absence of power, but writes are slow *and* cause significant wear.

The charge in a DRAM cell slowly leaks away. So each cell is read, and then written back to, several times a second by *refresh* circuitary to keep the contents stable. This is why this type of memory is called Dynamic RAM.

RAM: Random Access Memory – i.e. not block access (disk drive), nor sequential access (tape drive).

I recently read an article claiming that, in 1974, the Rutherford Appleton Laboratory added an extra 1MB to its main IBM 360/195. This cost about £400k then. Making some allowance for inflation, this is over £1 a byte.

DRAM in Detail



DRAM cells are arranged in (near-)square arrays. To read, first a row is selected and copied to a buffer, from which a column is selected, and the resulting single bit becomes the output. This example is a 64 bit DRAM.

This chip would need 3 *address lines* (i.e. pins) allowing 3 bits of address data to be presented at once, and a single `data line`. Also two pins for power, two for CAS and RAS, and one to indicate whether a read or a write is required.

Of course a 'real' DRAM chip would contain several tens of million bits.

DRAM Read Timings

To read a single bit from a DRAM chip, the following sequence takes place:

- Row placed on address lines, and Row Access Strobe pin signalled.
- After a suitable delay, column placed on address lines, and Column Access Strobe pin signalled.
- After another delay the one bit is ready for collection.
- The DRAM chip will automatically write the row back again, and will not accept a new row address until it has done so.

The same address lines are used for both the row and column access. This halves the number of addess lines needed, and adds the RAS and CAS pins.

Reading a DRAM cell causes a significant drain in the charge on its capacitor, so it needs to be refreshed before being read again.

More Speed!

The above procedure is tediously slow. However, for reading consecutive addresses, one important improvement can be made.

Having copied a whole row into the buffer (which is usually SRAM (see later)), if another bit from the same row is required, simply changing the column address whilst signalling the CAS pin is sufficient. There is no need to wait for the chip to write the row back, and then to rerequest the same row.

Today's SDRAM (Synchronous DRAM, including DDR, DDR2, DDR3, DDR4 etc.) takes this approach one stage further. It assumes that the next (several) bits are wanted, and sends them in sequence without waiting to receive requests for their column addresses.

The row in the output buffer is refered to as being 'open'. The buffer is typically around 1KB in size per chip, or around 8KB for a DIMM using eight chips in parallel.

Speed

Old-style memory quoted latencies which were simply the time it would take an idle chip to respond to a memory request. In the early 1980s this was about 250ns. By the early 1990s it was about 80ns.

Today timings are usually quoted in a form such as DDR4/2400 17-17-17.

DDR means Double Data Rate SDRAM, and 2400 is the speed (MHz) of that doubled data rate. The other figures are in clock-cycles of the undoubled rate (here 1200MHz, 0.833ns).

The first, T_{CL} or T_{CAS}, is the time to respond to a read if the correct row is already open (14.17ns).

The second, T_{RCD}, is the RAS to CAS delay. This is the time an idle chip takes to copy a row to its output buffer. So the latency for reading from an idle chip is 28.33ns.

The third, T_{RP}, Row Precharge, is the time required to write back the current row. This must be done even if the open row has only been read. So the latency for reading from a chip with the wrong row currently open is 42.5ns.

So in twenty years memory has got three times faster in terms of latency. And DDR3/1600 11-11-11 was faster than DDR4/2400 17-17-17.

Time matters to memory more than clock cycles, so the above module would probably run happily at 10-10-10-24, and possibly 9-9-9-23, if run at 1333MHz. Stating a T_{CAS} of 11 at 1600MHz means that the chip will respond in 13.75ns, but not in 12.5ns. Running at a T_{CAS} of 9 at 1333MHz requires a response in 13.5ns.

If the request was for a whole cache line (likely, see later), then the time to complete the request will be a further four clock cycles or 5ns (if a 64 byte cache line, served by a single DIMM delivering 64 bits (8 bytes) twice per clock cycle). If two DIMMs serve the request in parallel, this is reduced to two clock cycles (2.5ns), but, in practice, this is not what dual channel memory controllers usually do.

A Practical Example

```
# modprobe eeprom ; decode-dimms
---=== SPD EEPROM Information ===---
Fundamental Memory type                          DDR3 SDRAM
Module Type                                      UDIMM
---=== Memory Characteristics ===---
Maximum module speed                             1600 MHz (PC3-12800)
Size                                             4096 MB
tCL-tRCD-tRP-tRAS                                11-11-11-28
Supported CAS Latencies (tCL)                    11T, 10T, 9T, 8T, 7T, 6T
---=== Timings at Standard Speeds ===---
tCL-tRCD-tRP-tRAS as DDR3-1600                   11-11-11-28
tCL-tRCD-tRP-tRAS as DDR3-1333                   9-9-9-24
tCL-tRCD-tRP-tRAS as DDR3-1066                   7-7-7-19
tCL-tRCD-tRP-tRAS as DDR3-800                    6-6-6-14
---=== Timing Parameters ===---
Minimum Cycle Time (tCK)                         1.250 ns
Minimum CAS Latency Time (tAA)                   13.125 ns
Minimum Write Recovery time (tWR)                15.000 ns
Minimum RAS# to CAS# Delay (tRCD)                13.125 ns
Minimum Row Active to Row Active Delay (tRRD)    6.000 ns
Minimum Row Precharge Delay (tRP)                13.125 ns
Minimum Active to Precharge Delay (tRAS)         35.000 ns
---=== Optional Features ===---
Operable voltages                                1.5V
---=== Manufacturer Data ===---
Manufacturing Date                               2015-W21
```


An Older Example

```
---=== SPD EEPROM Information ===---
Fundamental Memory type                          DDR2 SDRAM
---=== Memory Characteristics ===---
Maximum module speed                             533 MHz (PC2-4200)
Size                                             1024 MB
Module Type                                      UDIMM (133.25 mm)
Voltage Interface Level                          SSTL 1.8V
tCL-tRCD-tRP-tRAS                                5-4-4-12
Supported CAS Latencies (tCL)                    5T, 4T, 3T
Minimum Cycle Time                               3.75 ns at CAS 5 (tCK min)
                                                 3.75 ns at CAS 4
                                                 5.00 ns at CAS 3
Maximum Cycle Time (tCK max)                     8.00 ns
---=== Timing Parameters ===---
Minimum Row Precharge Delay (tRP)                15.00 ns
Minimum RAS# to CAS# Delay (tRCD)                15.00 ns
Minimum RAS# Pulse Width (tRAS)                  45.00 ns
Write Recovery Time (tWR)                        15.00 ns
```

The older module is DDR2/533 made c. 2005, the newer DDR3/1600 made in 2015. In terms of latencies, T_{CAS} and T_{RCD} have improved from 15ns to 13.125ns, an improvement of well under 2% a year! If you think I should update these slides to show DDR4/2400 17-17-17, that is slower than the DDR3/1600 above.

Bandwidth

Bandwidth has improved much more over the same period. In the early 1980s memory was usually arranged to deliver 8 bits (one byte) at once, with eight chips working in parallel. By the early 1990s that had risen to 32 bits (4 bytes), and today one expects 128 bits (16 bytes) on any desktop.

More dramatic is the change in time taken to access consecutive items. In the 1980s the next item (whatever it was) took slightly longer to access, for the DRAM chip needed time to recover from the previous operation. So late 1980s 32 bit wide 80ns memory was very unlikely to deliver as much as four bytes every 100ns, or 40MB/s. Now sequential access is anticipated, and arrives at the doubled clock speed, so at 2400MHz for DDR4/2400 memory. Coupled with being arranged with 128 bits in parallel, this leads to a theoretical bandwidth of 38.4GB/s.

So in thirty years the bandwidth has improved by a factor of about 1000.

(SDRAM sustains this bandwidth by permitting new column access requests to be sent before the data from the previous are received. With a typical burst length of 8 bus transfers, or four cycles of the undoubled clock, new requests need to be sent every four clock cycles, yet T_{CAS} might be a dozen clock cycles.)

Note that terms such as ‘PCx-19200 memory’ refer to a 64 bit memory module capable of sustaining 19200MB/s, so DDRx/2400, or PC4-19200 here.

Parity & ECC

In the 1980s, business computers had memory arranged in bytes, with one extra bit per byte which stored parity information. This is simply the sum of the other bits, modulo 2. If the parity bit disagreed with the contents of the other eight bits, then the memory had suffered physical corruption, and the computer would usually crash, which is considered better than calmly going on generating wrong answers.

Calculating the parity value is quite cheap in terms of speed and complexity, and the extra storage needed is only 12.5%. However parity will detect only an odd number of bit-flips in the data protected by each parity bit. If an even number change, it does not notice. And it can never correct.

Better than parity is ECC memory (Error Correcting Code), usually SEC-DED (Single Error Corrected, Double Error Detected).

One common code for dealing with n bits requires an extra $2 + \log_2 n$ check bits. Each code now usually protects eight bytes, 64 bits, for which $2 + \log_2 64 = 8$ extra check bits are needed. Once more, 12.5% extra, or one extra bit per byte.

Causes and Prevelance of Errors

In the past most DRAM errors have been blamed on cosmic rays, more recent research suggest that this is not so. A study of Google's servers over a 30 month period suggests that faulty chips are a greater problem. Cosmic rays would be uniformly distributed, but the errors were much more clustered.

About 30% of the servers had at least one correctable error per year, but the average number of correctable errors per machine year was over 22,000. The probability of a machine which had one error having another within a year was 93%. The uncorrectable error rate was 1.3% per machine year.

The numbers are skewed by the fact that once insulation fails so as to lock a bit to one (or zero), then, on average, half the accesses will result in errors. In practice insulation can partially fail, such that the data are usually correct, unless neighbouring bits, temperature, . . . , conspire to cause trouble.

Uncorrectable errors were usually preceded by correctable ones: over 60% of uncorrectable errors had been preceded by a correctable error in the same DIMM in the same month, whereas a random DIMM has a less than 1% correctable error rate per month.

'DRAM Errors in the Wild: a Large-Scale Field Study', Schroeder *et al.*, Sigmetrics, pub 2009, for the data referenced in this slide.

ECC: Do We Care?

A typical home PC, run for a few hours each day, with only about half as much memory as those Google servers, is unlikely to see an error in its five year life. One has about a one in ten chance of being unlucky. When running a hundred machines 24/7, the chances of getting through a month, let alone a year, without a correctable error would seem to be low.

Intel's desktop i3/i5/i7 processors do not support ECC memory, whereas their server-class Xeon processors all do. Most major server manufacturers (HP, Dell, IBM, etc.) simply do not sell any servers without ECC. Indeed, most also support the more sophisticated 'Chipkill' correction which can cope with one whole chip failing on a bus of 128 data bits and 16 'parity' bits.

I have an 'ECC only' policy for servers, both file servers and machines likely to run jobs. In my Group, this means every desktop machine. The idea of doing financial calculations on a machine without ECC I find amusing and unauditable, but I realise that, in practice, it is what most Accounts Offices do. But money matters less than science.

Of course an undetected error may cause an immediate crash, it may cause results to be obviously wrong, it may cause results to be subtly wrong, or it may have no impact on the final result.

'Chipkill' is IBM's trademark for a technology which Intel calls Intel x4 SDDC (single device data correction). It starts by interleaving the bits to form four 36 bit words, each word having one bit from each chip, so a SEC-DED code is sufficient for each word.

Keeping up with the CPU

CPU clock speeds in the past twenty years have increased by a factor of around 500. (About 60MHz to about 3GHz.) Their performance in terms of instructions per second has increased by about 10,000, as now one generally has four cores, each capable of multiple instructions per clock cycle, not a single core struggling to maintain one instruction per clock cycle.

The partial answer is to use expensive, fast, cache RAM to store frequently accessed data. Cache is expensive because its SRAM uses multiple transistors per bit (typically six). It is fast, with sub-ns latency, lacking the output buffer of DRAM, and not penalising random access patterns.

But it is power-hungry, space-hungry, and needs to be physically very close to the CPU so that distance does not cause delay. $c = 1$ in units of feet per ns in vacuum. So a 3GHz signal which needs to travel just two inches and back again will lose a complete cycle. In silicon things are worse.

(Experimentalists claim that $c = 0.984$ft/ns.)

Expensive?

SRAM can be optimised for low power consumption, as its data-retention power can be very low: it has none of the dynamic refreshing that DRAM needs. So memory which requires a small lithium battery to retain data is probably SRAM. But here we are interested in speed-optimised memory.

Even finding power data for SDRAM DIMMs is quite hard, but I found a maximum value of 6.8W for a 16GB DDR3/1600 DIMM. So I conclude that the DRAM memory in a typical desktop is consuming around 10W.

The on-die CPU caches will take about half the die area, but probably only about 20% of the power, even when being active. This will again be around 10W for a desktop.

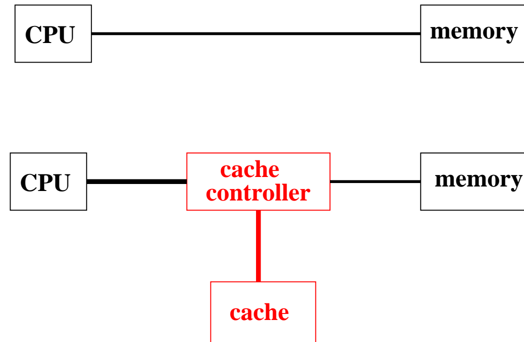
The big difference is that 10W has procured around 30GB of active DRAM, or about 6MB of active cache memory.

The above are very much order of magnitude estimates, but they suggest that the power requirements of active SRAM and DRAM are differing by a factor of a thousand or more. I am not sure I believe that it is quite that much, and much will depend on the SRAM speed, but it is clearly substantial.

Of course if power and money are no object, one can build a computer whose main memory is entirely SRAM. Thus the vector Crays and similar of the late 1980s and early 1990s. But eventually even Cray produced a 'budget' DRAM model (the J90).

Caches: the Theory

The theory of caching is very simple. Put a small amount of fast, expensive memory in a computer, and arrange automatically for that memory to store the data which are accessed frequently. One can then define a cache *hit rate*, that is, the number of memory accesses which go to the cache divided by the total number of memory accesses. This is usually expressed as a percentage & will depend on the code run.



The first paper to describe caches was published in 1965 by Maurice Wilkes (Cambridge). The first commercial computer to use a cache was the IBM 360/85 in 1968.

The Cache Controller

Conceptually this has a simple task:

- Intercept every memory request
- Determine whether cache holds requested data
- If so, read data from cache
- If not, read data from memory *and* place a copy in the cache as it goes past.

However, the second bullet point must be done *very* fast, and this leads to the compromises. A cache controller inevitably makes misses slower than they would have been in the absence of any cache, so to show a net speed-up hits have to be plentiful and fast. A badly designed cache controller can be worse than no cache at all.

An aside: Hex

A quick lesson in hexadecimal (base-16) arithmetic is due at this point. Computers use base-2, but humans tend not to like reading long base-2 numbers.

Humans also object to converting between base-2 and base-10. However, getting humans to work in base-16 and convert between base-2 and base-16 is easier.

Hex uses the letters A to F to represent the 'digits' 10 to 15. As $2^4 = 16$ conversion to and from binary is done trivially using groups of four digits.

```
0101 1101 0010 1010 1111 0001 1100 0011

  5    C    2    A    F    1    B    3
```

So

$$0101\,1101\,0010\,1010\,1111\,0001\,1100\,0011_2 = 5\mathrm{C2A\,F1B3}_{16}$$

As one hex digit is equivalent to four binary digits, two hex digits are exactly sufficient for one byte.

Hex numbers are often prefixed with '`0x`' to distinguish them from base ten.

When forced to work in binary, it is usual to group the digits in fours as above, for easy conversion into hex or bytes.

Our Computer

For the purposes of considering caches, let us consider a computer with a 1MB address space and a 64KB cache.

An address is therefore 20 bits long, or 5 hex digits, or $2\frac{1}{2}$ bytes.

Suppose we try to cache individual bytes. Each entry in the cache must store not only the data, but also the address in main memory it was taken from, called the *tag*. That way, the cache controller can look through all the tags and determine whether a particular byte is in the cache or not.

So we have 65536 single byte entries, each with a $2\frac{1}{2}$ byte tag.

A Disaster

This is bad on two counts.

A waste of space

We have 64KB of cache storing useful data, and 160KB storing tags.

A waste of time

We need to scan 65536 tags before we know whether something is in the cache or not. This will take far too long.

Lines

The solution to the space problem is not to track bytes, but *lines*. Consider a cache which deals in units of 16 bytes.

```
64KB = 65536 * 1 byte
     =  4096 * 16 bytes
```

We now need just 4096 tags.

Furthermore, each tag can be shorter. Consider a random address:

`0x23D17`

This can be read as byte `7` of line `23D1`. The cache will either have all of line `23D1` and be able to return byte number 7, or it will have none of it. Lines always start at an address which is a multiple of their length.

Decoding a byte address into a line and a byte within a line is instantaneous. In this case the bottom four bits represent the offset within a line, and the rest the line number. Hence the use of powers of two for everything.

Getting better…

A waste of space?

We now have 64KB storing useful data, and 8KB storing tags. Considerably better.

A waste of time

Scanning 4096 tags may be a 16-fold improvement, but is still a disaster.

Causing trouble

Because the cache can store only full lines, if the processor requests a single byte which the cache does not hold, the cache then requests the full line from the memory so that it can keep a copy of the line. Thus the memory might have to supply $16\times$ as much data as before! However, memory buses typically supply at least eight bytes at a time, and we know that SDRAM is very fast at providing a burst of multiple adjacent bytes.

A Further Compromise

We have 4096 lines, potentially addressable as line 0 to line 0xFFF.

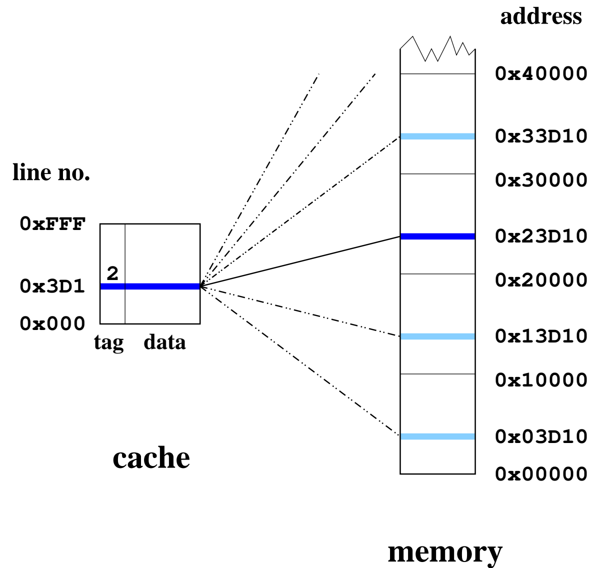
On seeing an address, e.g. `0x23D17`, we discard the last 4 bits, and scan all 4096 tags for the number `0x23D1`.

Why not always use line number `0x3D1` within the cache for storing this bit of memory? The advantage is clear: we need only look at one tag, and see if it holds the line we want, `0x23D1`, or one of the other 15 it could hold: `0x03D1, 0x13D1`, etc.

Indeed, the new-style tag need only hold that first hex digit, we know the other digits! This reduces the amount of tag memory to 2KB.

Direct Mapped Caches

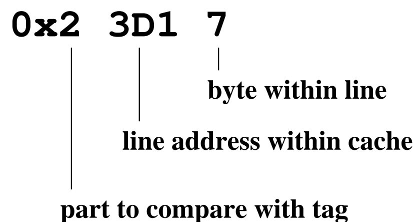
We have just developed a *direct mapped* cache. Each address in memory maps directly to a single location in cache, and each location in cache maps to multiple (here 16) locations in memory.



Success?

- The overhead for storing tags is 3%. Quite acceptable, and much better than 250%!
- Each 'hit' requires a tag to be looked up, a comparison to be made, and then the data to be fetched. Oh dear. This *tag RAM* had better be *very* fast.
- Each miss requires a tag to be looked up, a comparison to fail, and then a whole line to be fetched from main memory.
- The 'decoding' of an address into its various parts is instantaneous.

The zero-effort address decoding is an important feature of all cache schemes.



The Consequences of Compromise

At first glance we have done quite well. Any contiguous 64KB region of memory can be held in cache. (As long as it starts on a cache line boundary)

E.g. The 64KB region from `0x23840` to `0x3383F` would be held in cache lines `0x384` to `0xFFF` then `0x000` to `0x383`

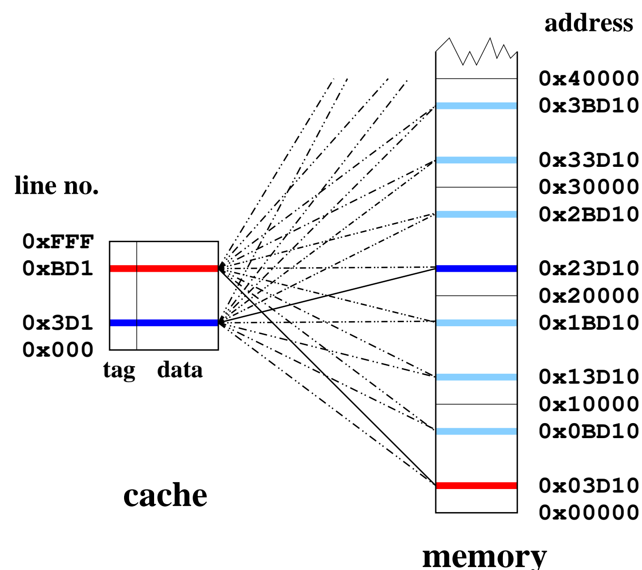
Even better, widely separated pieces of memory can be in cache simultaneously. E.g. `0x15674` in line `0x567` and `0xC4288` in line `0x428`.

However, consider trying to cache the two bytes `0x03D11` and `0x23D19`. This cannot be done: both map to line `0x3D1` within the cache, but one requires the memory area from `0x03D10` to be held there, the other the area from `0x23D10`.

Repeated accesses to these two bytes would cause cache *thrashing*, as the cache repeatedly caches then throws out the same two pieces of data.

Associativity

Rather than each line in memory being storable in just one location in cache, why not make it two?



Thus a *2-way associative* cache, which requires two tags to be inspected for every access & an extra bit per tag. Can generalise to 2^n-way associativity.

Anti Thrashing Entries

Anti Thrashing Entries are a cheap way of increasing the effective associativity of a cache for simple cases. One extra cache line, complete with tag (including the old line address within the cache), is stored, and it contains the last line expelled from the cache proper.

This line is checked for a 'hit' in parallel with the rest of the cache, and if a hit occurs, it is moved back into the main cache, and the line it replaces is moved into the ATE.

Some caches have several ATEs, rather than just one.

```
double precision a(2048,2),x
do i=1,2048
  x=x+a(i,1)*a(i,2)
enddo
```

```
double a[2][2048],x;
for(i=0;i<2047;i++){
  x+=a[0][i]*a[1][i];
}
```

Assume a 16K direct mapped cache with 32 byte lines. `a(1,1)` comes into cache, pulling `a(2-4,1)` with it. Then `a(1,2)` displaces all these, at it must be stored in the same line, as its address modulo 16K is the same. So a(2,1) is not found in cache when it is referenced. With a single ATE, the cache hit rate jumps from 0% to 75%, the same that a 2-way associative cache would have achieved for this algorithm.

Remember that Fortran and C store arrays in the opposite order in memory. Fortran will have `a(1,1)`, `a(2,1)`, `a(3,1)`..., whereas C will have `a[0][0]`, `a[0][1]`, `a[0][2]`...

A Hierarchy

The speed gap between main memory and the CPU core is so great that there are usually multiple levels of cache.

The first level, or *primary cache*, is small (typically 16KB to 128KB), physically attached to the CPU, and runs as fast as possible, which implies low associativity.

The next level, or *secondary cache*, is larger (typically 256KB to 8MB), slower, and has a higher associativity. There may even be a third level too.

Typical times in clock-cycles to serve a memory request would be:

primary cache	2-4
secondary cache	5-25
main memory	100-400

Cf. functional unit speeds on page 34.

Write Back or Write Through?

Should data written by the CPU modify merely the cache if those data are currently held in cache, or modify the memory too? The former, *write back*, can be faster, but the latter, *write through*, is simpler.

With a write through cache, the definitive copy of data is in the main memory. If something other than the CPU (e.g. a disk controller or a second CPU) writes directly to memory, the cache controller must *snoop* this traffic, and, if it also has those data in its cache, update (or invalidate) the cache line too.

Write back caches add two problems. Firstly, anything else reading directly from main memory must have its read intercepted if the cached data for that address differ from the data in main memory.

Secondly, on ejecting an old line from the cache to make room for a new one, if the old line has been modified it must first be written back to memory, for it has the current data and the copy in memory is out of date.

Each cache line therefore has an extra bit in its tag, which records whether the line is modified, or *dirty*.

Cache Design Decision

If a write is a miss, should the cache line be filled (as it would for a read)? If the data just written are read again soon afterwards, filling is beneficial, as it is if a write to the same line is about to occur. However, caches which allocate on writes perform badly on randomly scattered writes. Each write of one word is converted into *reading* the cache line from memory, modifying the word written in cache and marking the whole line dirty. When the line needs discarding, the whole line will be written to memory. Thus writing one word has been turned into two lines worth of memory traffic.

What line size should be used? What associativity?

If a cache is n-way associative, which of the n possible lines should be discarded to make way for a new line? A random line? The least recently used? A random line excluding the most recently used?

As should now be clear, not all caches are equal!

The 'random line excluding the most recently used' replacement algorithm (also called pseudo-LRU) is easy to implement. One bit marks the most recently used line of the associative set. True LRU is harder (except for 2-way associative).

Not All Data are Equal

If the cache controller is closely associated with the CPU, it can distinguish memory requests from the instruction fetcher from those from the load/store units. Thus instructions and data can be cached separately.

This almost universal *Harvard Architecture* prevents poor data access patterns leaving both data and program uncached. However, usually only the first level of cache is split in this fashion.

The instruction cache is usually write-through, whereas the data cache is usually write-back. Write-through caches never contain the 'master' copy of any data, so they can be protected by simple parity bits, and the master copy reloaded on error. Write back caches ought to be protected by some form of ECC, for if they suffer an error, they may have the only copy of the data now corrupted.

The term 'Harvard architecture' comes from an early American computer which used physically separate areas of main memory for storing data and instructions. No modern computer does this.

Explicit Prefetching

One spin-off from caching is the possibility of *prefetching*.

Many processors have an instruction which requests that data be moved from main memory to primary cache when it is next convenient.

If such an instruction is issued ahead of some data being required by the CPU core, then the data may have been moved to the primary cache by the time the CPU core actually wants them. If so, much faster access results. If not, it doesn't matter.

If the latency to main memory is 100 clock cycles, the prefetch instruction ideally needs issuing 100 cycles in advance, and many tens of prefetches might be busily fetching simultaneously. Most current processors can handle a couple of simultaneous prefetches…

Implicit Prefetching

Some memory controllers are capable of spotting certain access patterns as a program runs, and prefetching data automatically. Such prefetching is often called *streaming*.

The degree to which patterns can be spotted varies. Unit stride is easy, as is unit stride backwards. Spotting different simultaneous streams is also essential, as a simple dot product:

```
do i=1,n
  d=d+a(i)*b(i)
enddo
```

leads to alternate unit-stride accesses for `a` and `b`.

IBM's Power3 processor (1998), and Intel's Pentium 4 (2000) both spotted simple patterns in this way, as do most modern CPUs. Unlike software prefetching, no support from the compiler is required, and no instructions exist to make the code larger and occupy the instruction decoder. However, streaming is less flexible.

Clock multiplying

Today all of the caches are usually found on the CPU die, rather than on external chips. Whilst the CPU is achieving hits on its caches, it is unaffected by the slow speed of the outside world (e.g. main memory).

Thus it makes sense for the CPU internally to use much higher clock-speeds than its external bus. The gap is actually decreasing currently as CPU speeds are levelling off at around 3GHz, whereas external bus speeds are continuing to rise. In former days the gap could be very large, such as the last of the Pentium IIIs which ran at around 1GHz internally, with a 133MHz external bus. In the days when caches were external to the CPU on the motherboard there was very little point in the CPU running faster than its bus. Now it works well provided that the cache hit rate is high (>90%), which will depend on both the cache architecture and the program being run.

In order to reduce power usage, not all of the CPU die uses the same clock frequency. It is common for the last level cache, which is responsible for around half the area of the die, to use clock speeds which are only around a half or a third of those of the CPU core and the primary cache.

Thermal Limits to Clock Multiplying

The rate at which the transistors which make up a CPU switch is controlled by the rate at which carriers get driven out of their gate regions. For a given chip, increasing the electric field, i.e. increasing the voltage, will increase this speed. Until the voltage is so high that the insulation fails.

The heat generated by a CPU contains both a simple ohmic term, arising from leakage across the narrow insulating barriers, proportional to the square of the voltage, and a term from the charging of capacitors through a resistor (modelling the change in state of data lines and transistors). This is proportional to frequency, the square of the voltage, and the proportion, α, of the chip which is actively changing state.

Once the CPU gets too hot, thermally excited carriers begin to swamp the intrinsic carriers introduced by the n and p doping. With the low band-gap of silicon, the maximum junction temperature is around $90°$C, or just $50°$C above the air temperature which most computers can allegedly survive.

Current techniques allow around 120W to be dissipated from a chip with forced air cooling.

$$P = V^2 \left(\frac{1}{R} + \nu \alpha C \right)$$

$$V = V_0 + a\nu^2$$

Smaller is Better?

As one decreases the feature size on a chip, by moving to a yet more modern fabrication technique, C, R and a all decrease. The decrease in R is a problem. Whereas historically the ohmic term above was pretty negligible, today in some operating circumstances it can be responsible for around half of the power drawn by a CPU.

Around 1990 a typical feature size on a CPU was 1μm. Now it is as little as 14nm. The obvious result of this is that a modern CPU contains a lot more transistors than one from the 1990s. The number of transistors per unit area does not quite scale as the inverse square of the feature size, but a 14nm Skylake CPU has over a thousand times as many transistors as a 1μm 486DX, and it occupies $1.5\times$ the area (about 1cm^2).

Laptops, and the more modern desktops, have power-saving modes in which the clock speed is first dropped, and then a fraction of a second later, the supply voltage also dropped. The GPU in graphics cards usually offers similar power-saving features.

Other tricks involve removing the clock signal from idle parts of a CPU, or even removing the supply voltage entirely. This is simplest with multi-core CPUs. Rather than keep idle cores active at a reduced clock frequency, some designs allow for them to be turned off entirely.

The Relevance of Theory

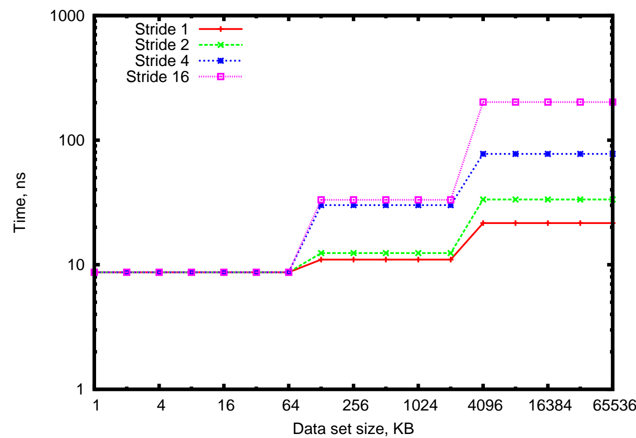
```
integer a(*),i,j                int i,j,*a;

j=1                             j=1;
do i=1,n                        for (i=0;i<n;i++){
  j=a(j)                          j=a[j];
enddo                           }
```

This code is mad. Every iteration depends on the previous one, and significant optimisation is impossible.

However, the memory access pattern can be changed dramatically by changing the contents of `a`. Setting `a(i)=i+1` and `a(k)=1` will give consecutive accesses repeating over the first `k` elements, whereas `a(i)=i+2`, `a(k-1)=2` and `a(k)=1` will access alternate elements, etc.

One can also try pseudorandom access patterns. They tend to be as bad as large stride access.

Classic caches

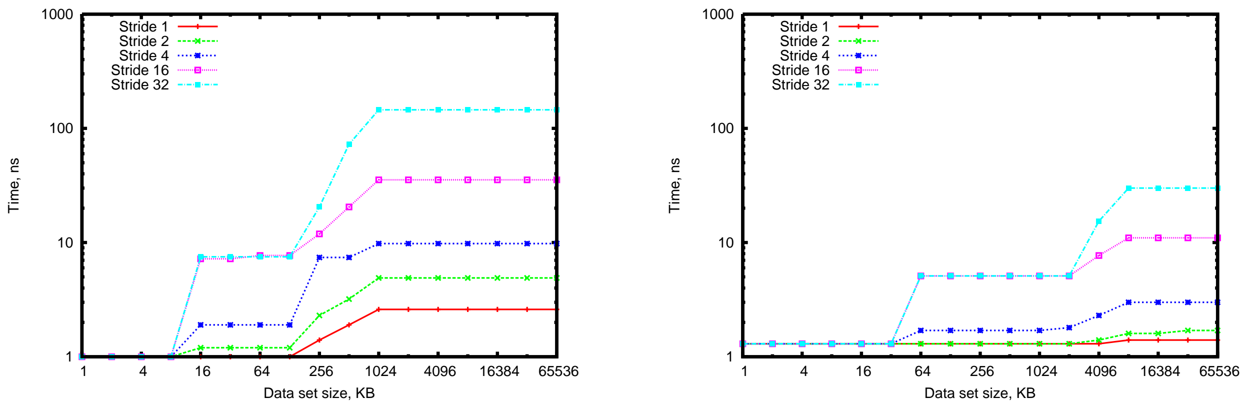


With a 16 element (64 bytes) stride, we see access times of 8.7ns for primary cache, 33ns for secondary, and 202ns for main memory. The cache sizes are clearly 64KB and 2MB.

With a 1 element (4 bytes) stride, the secondary cache and main memory appear to be faster. This is because once a cache line has been fetched from memory, the next 15 accesses will be primary cache hits on the next elements of that line. The average should be $(15 * 8.7 + 202)/16 = 20.7$ns, and 21.6ns is observed.

The computer used for this was a 463MHz XP900 (Alpha 21264). It has 64 byte cache lines.

Performance Enhancement



On the left a 2.4GHz Pentium 4 (launched 2002, RAMBUS memory), and on the right a 2.4GHz Core 2 quad core (launched 2008, DDR3 memory). Both have 64 byte cache lines.

For the Pentium 4, the fast 8KB primary cache is clearly seen, and a 512KB secondary less clearly so. The factor of four difference between the main memory's latency at a 64 byte and 128 byte stride is caused by automatic hardware prefetching into the secondary cache. For strides of up to 64 bytes inclusive, the hardware notices the memory access pattern, even though it is hidden at the software level, and starts fetching data in advance automatically.

For the Core 2 the caches are larger – 32KB and 4MB, and the main memory is a little faster. But six years and three generations of memory technology have changed remarkably little.

Matrix Multiplication: $A_{ij} = B_{ik}C_{kj}$

```
do i=1,n
  do j=1,n
    t=0
    do k=1,n
      t=t+b(i,k)*c(k,j)
    enddo
    a(i,j)=t
  enddo
enddo
```

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

The above Fortran has unit stride access on the array `c` in the inner loop, but a stride of `n` doubles on the array `b`. The C manages unit stride on `b` and a stride of `n` doubles on the array `c`. Neither manages unit stride on both arrays.

Optimising this is not completely trivial, but is very worthwhile.

Very Worthwhile

The above code running on a 2.4GHz Core 2 managed around 500 MFLOPS at a matrix size of 64, dropping to 115 MFLOPS for a matrix size of 1024.

Using an optimised linear algebra library increased the speed for the smaller sizes to around 4,000 MFLOPS, and for the larger sizes to around 8,700 MFLOPS, close to the computer's peak speed of 9,600 MFLOPS.

There are many possibilities to consider for optimising this code. If the matrix size is very small, don't, for it will all fit in L1 cache anyway. For large matrices one can consider transposing the matrix which would otherwise be accessed with the large stride. This is most beneficial if that matrix can then be discarded (or, better, generated in the transposed form). Otherwise one tries to modify the access pattern with tricks such as

```
do i=1,nn,2
  do j=1,nn
    t1=0 ; t2=0
    do k=1,nn
      t1=t1+b(i,k)*c(k,j)       ! Remember that b(i,k) and
      t2=t2+b(i+1,k)*c(k,j)     ! b(i+1,k) are adjacent in memory
    enddo
    a(i,j)=t1
    a(i+1,j)=t2
  enddo
enddo
```

This halves the number of passes through `b` with the large stride, and therefore shows an immediate doubling of speed at `n=1024` from 115 MFLOPS to 230 MFLOPS. Much more to be done before one reaches 8,000 MFLOPS though, so don't bother: link with a good BLAS library and use its matrix multiplication routine! (Or use the F90 intrinsic `matmul` function in this case.) [If trying this at home, note that many Fortran compilers spot simple examples of matrix multiplication and re-arrange the loops themselves. This can cause confusion.]

Memory Access Patterns in Practice

Matrix Multiplication

We have just seen that very different speeds of execution can be obtained by different methods of matrix multiplication.

Matrix multiplication is not only quite a common problem, but it is also very useful as an example, as it is easy to understand and reveals most of the issues.

More Matrix Multiplication

$$A_{ij} = \sum_{k=1,N} B_{ik} C_{kj}$$

So to form the product of two $N \times N$ square matrices takes N^3 multiplications and N^3 additions (less about N^2 additions, which is of little consequence).

The amount of memory occupied by the matrices scales as N^2, and is exactly $24N^2$ bytes assuming all are distinct and double precision.

Most of these examples use $N = 2048$, so require around 100MB of memory, and will take 16s if run at 1 GFLOPs.

(We ignore methods such as Strassen's algorithm, and similar, which scale slightly better at the cost of more memory, more complexity, and working best with particular sizes, such as $N = 2^n$. Strassen produces multiplication in order $N^{2.807}$ operations ($\log_2 7$ is the exact result for 2.807). Its prefactor is 6, not 2, so the cross-over point in operation count is at about $N = 300$. It also has twice as many additions as multiplications, which makes less efficient use of most modern CPUs, and moves the cross-over point to over 1,300.)

Our Computer

These examples use a 2.4GHz quad core Core2 with 4GB of RAM. Each core can complete two additions and two multiplications per clock cycle, so its theoretical sustained performance is 9.6 GFLOPs.

Measured memory bandwidth for unit stride access over an array of 64MB is 6GB/s, and for access with a stride of 2048 doubles it is 84MB/s (one item every 95ns).

We will also consider something older and simpler, a 2.8GHz 32 bit Pentium 4 with 3GB of RAM. Theoretical sustained performance is 5.6 GFLOPs, 4.2GB/s and 104ns. Its data in the following slides will be shown in italics in square brackets.

And also something newer: a 2.5GHz quad core Haswell: theoretical sustained performance 40 GFLOPS per core. Its memory bandwidth is 20GB/s, but with a large stride one sees a latency of 74ns, so 108MB/s.

The Core 2 processor used, a Q6600, was first released in 2007. The Pentium 4 used was first released in 2002. The successor to the Core 2, the Nehalem, was first released late in 2008. After the Sandy/Ivy Bridge generation, the Haswell followed in 2013.

Speeds

```
do i=1,n                        for(i=0;i<n;i++){
  do j=1,n                        for(j=0;j<n;j++){
    t=0                             t=0;
    do k=1,n                        for(k=0;k<n;k++){
      t=t+b(i,k)*c(k,j)               t+=b[i][k]*c[k][j];
    enddo                           }
    a(i,j)=t                        a[i][j]=t;
  enddo                           }
enddo                           }
```

If the inner loop is constrained by the compute power of the processor, it will achieve 9.6 GFLOPs. *[P4: 5.6 GFLOPS, Haswell: 40 GFLOPS]*

If constrained by bandwidth, loading two doubles and performing two FLOPS per iteration, it will achieve 750 MFLOPs. *[P4: 520 MFLOPS, Haswell: 2500 MFLOPS]*

If constrained by the large stride access, it will achieve two FLOPs every 95ns, or 21 MFLOPs. *[P4: 19 MFLOPS, Haswell: 27 MFLOPS]*

The First Result

When compiled with `gfortran -O0` the code achieved 41.6 MFLOPS. *[P4: 37 MFLOPS, Haswell: 111 MFLOPS]*

The code could barely be less optimal – even `t` was written out to memory, and read in from memory, on each iteration. The processor has done an excellent job with the code to achieve 47ns per iteration of the inner loop. This must be the result of some degree of speculative loading overlapping the expected 95ns latency.

In the mess which follows, one can readily identify the memory location `-40(%rbp)` with `t`, and one can also see two integer multiplies as the offsets of the elements `b(i,k)` and `c(k,j)` are calculated.

Messy

```
.L22:
        movq    -192(%rbp), %rbx
        movl    -20(%rbp), %esi
        movslq  %esi, %rdi
        movl    -28(%rbp), %esi
        movslq  %esi, %r8
        movq    -144(%rbp), %rsi
        imulq   %r8, %rsi
        addq    %rsi, %rdi
        movq    -184(%rbp), %rsi
        leaq    (%rdi,%rsi), %rsi
        movsd   (%rbx,%rsi,8), %xmm1
        movq    -272(%rbp), %rbx
        movl    -28(%rbp), %esi
        movslq  %esi, %rdi
        movl    -24(%rbp), %esi
        movslq  %esi, %r8
        movq    -224(%rbp), %rsi
        imulq   %r8, %rsi
        addq    %rsi, %rdi
        movq    -264(%rbp), %rsi
        leaq    (%rdi,%rsi), %rsi
        movsd   (%rbx,%rsi,8), %xmm0
        mulsd   %xmm1, %xmm0
        movsd   -40(%rbp), %xmm1
        addsd   %xmm1, %xmm0
        movsd   %xmm0, -40(%rbp)
        cmpl    %ecx, -28(%rbp)
        sete    %bl
        movzbl  %bl, %ebx
        addl    $1, -28(%rbp)
        testl   %ebx, %ebx
        je      .L22
```


Faster

When compiled with `gfortran -O1` the code achieved 118 MFLOPS. The much simpler code produced by the compiler has given the processor greater scope for speculation and simultaneous outstanding memory requests. The Haswell also improved significantly, reaching 180 MFLOPS. Don't expect older (or more conservative) processors to be this smart – on an ancient Pentium 4 the speed improved from 37.5 MFLOPS to 37.7 MFLOPS.

Notice that `t` is now maintained in a register, `%xmm0`, and not written out to memory on each iteration. The integer multiplications of the previous code have all disappeared, one by conversion into a Shift Arithmetic Left Quadbyte of 11 (i.e. multiply by 2048, or `2^11`).

```
.L10:
        movslq  %eax, %rdx
        movq    %rdx, %rcx
        salq    $11, %rcx
        leaq    -2049(%rcx,%r8), %rcx
        addq    %rdi, %rdx
        movsd   0(%rbp,%rcx,8), %xmm1
        mulsd   (%rbx,%rdx,8), %xmm1
        addsd   %xmm1, %xmm0
        addl    $1, %eax
        leal    -1(%rax), %edx
        cmpl    %esi, %edx
        jne     .L10
```

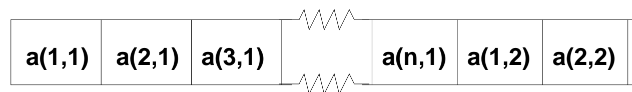
Unrolling: not faster

```
do i=1,nn
  do j=1,nn
    t=0
    do k=1,nn,2
      t=t+b(i,k)*c(k,j)+b(i,k+1)*c(k+1,j)
    enddo
    a(i,j)=t
  enddo
enddo
```

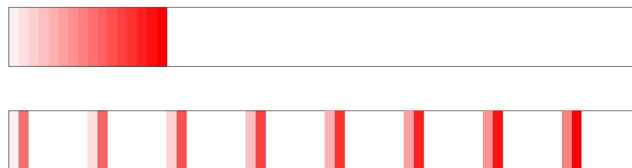
This ‘optimisation’ reduces the overhead of testing the loop exit condition, and little else. The memory access pattern is unchanged, and the speed is also pretty much unchanged – up by about 4%.

Memory Access Pattern

Fortran data storage order:



Below an 8×8 array being accessed the correct and incorrect way around.



Blocking: Faster

```
do i=1,nn,2
  do j=1,nn
    t1=0
    t2=0
    do k=1,nn
      t1=t1+b(i,k)*c(k,j)
      t2=t2+b(i+1,k)*c(k,j)
    enddo
    a(i,j)=t1
    a(i+1,j)=t2
  enddo
enddo
```

This has changed the memory access pattern on the array `b`. Rather than the pessimal order
`b(1,1) b(1,2) b(1,3) b(1,4) ... b(1,n) b(2,1) b(2,2)`
we now have
`b(1,1) b(2,1) b(1,2) b(2,2) ..... b(1,n) b(2,n) b(3,1) b(4,1)`
Every other item is fetched almost for free, because its immediate neighbour has just been fetched. The number of iterations within this inner loop is the same, but the loop is now executed half as many times.

Yes, Faster

We would predict a speedup of about a factor of two, and that is indeed seen. Now the Core 2 reaches 203 MFLOPS (up from 118 MFLOPS), the Pentium 4 71 MFLOPS (up from 38 MFLOPS), and the Haswell 375 MFLOPS (up from 180 MFLOPS).

Surprisingly changing the blocking factor from 2 to 4 (i.e. four elements calculated in the inner loop) did not impress the Core 2. It improved to just 224 MFLOPS (+10%). The Pentium 4, which had been playing fewer clever tricks in its memory controller, was much happier to see the blocking factor raised to 4, now achieving 113 MFLOPS (+59%), and the Haswell also was happier, managing 520 MFLOPS (+38%).

Note we have now got the Pentium 4 up to the speed we saw on our first test for the Haswell.

More, more more!

```
do i=1,nn,nb
  do j=1,nn
    do kk=0,nb-1
      a(i+kk,j)=0
    enddo
    do k=1,nn
      do kk=0,nb-1
        a(i+kk,j)=a(i+kk,j)+b(i+kk,k)*c(k,j)
      enddo
    enddo
  enddo
enddo
```

With `nb=1` this code is mostly equivalent to our original naïve code. Only less readable, potentially buggier, more awkward for the compiler, and `a(i,j)` is now unlikely to be cached in a register. With `nb=1` the Core 2 achieves 74 MFLOPS, the Pentium 4 33 MFLOPS, and the Haswell 176 MFLOPS. But with `nb=64` the Core 2 achieves 530 MFLOPS, the Pentium 4 320 MFLOPS and the Haswell 670 MFLOPS – their best scores so far.

Better, better, better

```
do k=1,nn,2
  do kk=0,nb-1
    a(i+kk,j)=a(i+kk,j)+b(i+kk,k)*c(k,j)+   &
                        b(i+kk,k+1)*c(k+1,j)
  enddo
enddo
```

Fewer loads and stores on `a(i,j)`, and the Core 2 likes this, getting 707 MFLOPS. The Pentium 4 now manages 421 MFLOPS, and the Haswell 1000 MFLOPS. Again this is trivially extended to a step of four in the `k` loop, which achieves 750 MFLOPS *[P4: 448 MFLOPS, Haswell: 1250 MFLOPS]*

Other Orders

```
a=0
do j=1,nn
  do k=1,nn
    do i=1,nn
      a(i,j)=a(i,j)+b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

Much better. 1 GFLOPS on the Core 2, 660 MFLOPS on the Pentium 4, and 1,600 GFLOPS on the Haswell.

In the inner loop, `c(k,j)` is constant, and so we have two loads and one store, all unit stride, with one add and one multiply.

Better Yet

```
a=0
do j=1,nn,2
  do k=1,nn
    do i=1,nn
      a(i,j)=a(i,j)+b(i,k)*c(k,j)+ &
                    b(i,k)*c(k,j+1)
    enddo
  enddo
enddo
```

Now the inner loop has `c(k,j)` and `c(k,j+1)` constant, so still has two loads and one store, all unit stride (assuming efficient use of registers), but now has two adds and two multiplies.

All processors love this – 1.48 GFLOPS on the Core 2, 1.21 GFLOPS on the Pentium 4, 2.7 GFLOPS on the Haswell.

Limits and Spills

Should we exend this by another factor of two, and make the outer loop of step 4?

The Core 2 says a clear yes, improving to 1.93 GFLOPS (+30%). The Pentium 4 is less enthusiastic, improving to 1.36 GFLOPS (+12%). The Haswell also gives a clear yes, improving to 3.5 GFLOPS (+28%)

What about 8? The Core 2 then gives 2.33 GFLOPS (+20%), the Pentium 4 1.45 GFLOPS (+6.6%), and the Haswell 3.90 GFLOPS (+11%).

With a step of eight in the outer loop, there are eight constants in the inner loop, `c(k,j)` to `c(k,j+7)`, as well as the two variables `a(i,j)` and `b(i,k)`. The Pentium 4 has just run out of registers, so three of the constant `c`'s have to be loaded from memory (cache) as they don't fit into registers.

The Core 2 has twice as many FP registers, so has not suffered what is called a 'register spill', when values which ideally would be kept in registers spill back into memory as the compiler runs out of register space.

Horrid!

Are the above examples correct? Probably not – I did not bother to test them!

The concepts are correct, but the room for error in coding in the above style is large. Also the above examples assume that the matrix size is divisible by the block size. General code needs (nasty) sections for tidying up when this is not the case.

Also, we are achieving around 20% of the peak performance of the processor. Better than the initial 1-2%, but hardly wonderful.

Best Practice

Be lazy. Use someone else's well-tested code where possible.

Using Intel's Maths Kernel Library one achieves 4.67 GFLOPS on the Pentium 4, 8.88 GFLOPS on one core of a Core 2, and 18.9 GFLOPS on the Haswell. Better, that library can make use of multiple cores of the Core 2 with no further effort, then achieving 33.75 GFLOPS when using all four cores.

The Haswell was confused by the idea of using all four cores, with speeds varying from 23 GFLOPS to 83 GFLOPS. Increasing the matrix size to 4096 produced much better consistency, and numbers of between 109 and 113 GFLOPS.

N.B.

```
call cpu_time(time1)
...
call cpu_time(time2)
write(*,*) time2-time1
```

records total CPU time, so does not show things going faster as more cores are used. One wants wall-clock time:

```
call system_clock(it1,ic)
time1=real(it1,kind(1d0))/ic
...
```


Other Practices

Use Fortran90's `matmul` routine.

Core 2

ifort -O3:	5.10 GFLOPS
gfortran:	3.05 GFLOPS
pathf90 -Ofast:	2.30 GFLOPS
pathf90	1.61 GFLOPS
ifort:	0.65 GFLOPS

Pentium 4

ifort -O3:	1.55 GFLOPS
gfortran:	1.05 GFLOPS
ifort:	0.43 GFLOPS

Lessons

Beating the best professional routines is hard.

Beating the worst isn't.

The variation in performance due to the use of different routines is *much* greater than that due to the single-core performance difference between a Pentium 4 and a Haswell. Indeed, the Pentium 4's best result is over $10\times$ as fast as the Haswell's worst result.

Difficulties

For the hand-coded tests, the original naïve code on slide 93 compiled with `gfortran -O1` recorded 118 MFLOPS *[37.7 MFLOPS]*, and was firmly beaten by reversing the loop order (slide 100) at 1 GFLOPS *[660 MFLOPS]*.

Suppose we re-run these examples with a matrix size of 25×25 rather than 2048×2048. Now the speeds are 1366 MFLOPS *[974 MFLOPS]* and 1270 MFLOPS *[770 MFLOPS]*.

The three arrays take $3 \times 25 \times 25 \times 8$ bytes, or 15KB, so things fit into L1 cache on both processors. L1 cache is insensitive to data access order, but the 'naïve' method allows a cache access to be converted into a register access (in which a sum is accumulated).

Leave it to Others!

So comparing these two methods, on the Core 2 the one which wins by a factor of 8.5 for the large size is 7% slower for the small size. For the Pentium 4 the results are more extreme: 17× faster for the large case, 20% slower for the small case.

A decent matrix multiplication library will use different methods for different problem sizes, ideally swapping between them at the precisely optimal point. It is also likely that there will be more than two methods used as one moves from very small to very large problems.

Reductio ad Absurdum

Suppose we now try a matrix size of 2×2. The 'naïve' code now manages 400 MFLOPS *[540 MFLOPS]*, and the reversed code 390 MFLOPS *[315 MFLOPS]*.

If instead one writes out all four expressions for the elements of `a` explicitly, the speed jumps to about 3,200 MFLOPS *[1,700 MFLOPS]*.

Loops of unknown (at compile time) but small (at run time) iteration count can be quite costly compared to the same code with the loop entirely eliminated.

For the first test, the 32 bit compiler really did produce significantly better code than the 64 bit compiler, allowing the Pentium 4 to beat the Core 2.

Maintaining Zero GFLOPS

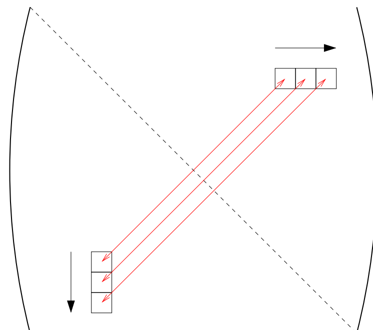
One matrix operation for which one can never exceed zero GFLOPS is the transpose. There are no floating point operations, but the operation still takes time.

```
do i=1,nn
  do j=i+1,nn
    t=a(i,j)
    a(i,j)=a(j,i)
    a(j,i)=t
  enddo
enddo
```

This takes about 24ns per element in `a` on the Core 2 *[P4: 96ns, Haswell: 12ns]* with a matrix size of 4096.

Problems

It is easy to see what is causing trouble here. Whereas one of the accesses in the loop is sequential, the other is of stride 32K. We would naïvely predict that this code would take around 43ns *[52ns]* per element, based on one access taking negligible time, and the other the full latency of main memory.

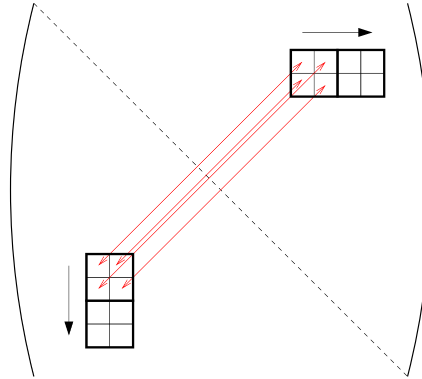


The Pentium 4 is doing worse than our naïve model because 104ns is its access time for *reads* from main memory. Here we have writes as well, so there is a constant need to evict dirty cache lines. This will make things worse.

The Core 2 is showing the sophistication of a memory controller capable of having several outstanding requests and a CPU capable of speculation.

Faster

If the inner loop instead dealt with a small 2×2 block of element, it would have two stride 32K accesses per iteration and exchange eight elements, instead of one stride 32K access to exchange two elements. If the nasty stride is the problem, this should run twice as fast. It does: 12ns per element *[42ns]*.



Nasty Code

```
  do i=1,nn,2
    do j=i+2,nn,2
      t=a(i,j)
      a(i,j)=a(j,i)
      a(j,i)=t
      t=a(i+1,j)
      a(i+1,j)=a(j,i+1)
      a(j,i+1)=t
      t=a(i,j+1)
      a(i,j+1)=a(j+1,i)
      a(j+1,i)=t
      t=a(i+1,j+1)
      a(i+1,j+1)=a(j+1,i+1)
      a(j+1,i+1)=t
    enddo
  enddo

  do i=1,nn,2
    j=i+1
    t=a(i,j)
    a(i,j)=a(j,i)
    a(j,i)=t
 enddo
```

Is this even correct? Goodness knows – it is unreadable, and untested. And it is certainly wrong if `nn` is odd.

How far should we go?

Why not use a 3×3 block, or a 10×10 block, or some other $n \times n$ block? For optimum speed one should use a larger block than 2×2.

Ideally we would read in a whole cache line and modify all of it for the sequential part of reading in a block in the lower left of the matrix. Of course, we can't. There is no guarantee that the array starts on a cache line boundary, and certainly no guarantee that each row starts on a cache line boundary.

We also want the whole of the block in the upper right of the matrix to stay in cache whilst we work on it. Not usually a problem – level one cache can hold a couple of thousand doubles, *but* with a matrix size which is a large power of two, `a(i,j)` and `a(i,j+1)` will be separated by a multiple of the cache size, and in a direct mapped cache will be stored in the same cache line.

Different Block Sizes

Block Size	Pentium 4	Athlon II	Core 2	Haswell
1	100ns	41ns	25ns	10ns
2	42ns	22ns	12ns	5ns
4	27ns	21ns	11ns	5ns
8	22ns	19ns	8ns	5ns
16	64ns	17ns	8ns	5ns
32	88ns	41ns	9ns	10ns
64	102ns	41ns	12ns	10ns

Caches:

Pentium 4:	L1 16K 4 way, L2 512K 8 way.
Athlon II:	L1 64K 2 way, L2 1MB 16 way.
Core 2:	L1 32K 8 way, L2 4MB 16 way.
Haswell:	L1 32K 8 way, L2 256K 8 way, L3 8MB 16 way

Notice that even on this simple test we have the liberty of saying that the Athlon II is merely 15% faster than the old Pentium 4, or a more respectable $3.75\times$ faster. One can prove almost anything with benchmarks. I have several in which that Athlon II would easily beat that Core 2…

Nastier Code

```
do i=1,nn,nb
  do j=i+nb,nn,nb
    do ii=0,nb-1
      do jj=0,nb-1
        t=a(i+ii,j+jj)
        a(i+ii,j+jj)=a(j+jj,i+ii)
        a(j+jj,i+ii)=t
      enddo
    enddo
  enddo
enddo

do i=1,nn,nb
  j=i
  do ii=0,nb-1
    do jj=ii+1,nb-1
      t=a(i+ii,j+jj)
      a(i+ii,j+jj)=a(j+jj,i+ii)
      a(j+jj,i+ii)=t
    enddo
  enddo
enddo
```

Is this even correct? Goodness knows – it is unreadable, and untested. And it is certainly wrong if `nn` is not divisible by `nb`.

Different Approaches

One can also transpose a square matrix by recursion: divide the matrix into four smaller square submatrices, transpose the two on the diagonal, and transpose and exchange the two off-diagonal submatrices.

For computers which like predictable strides, but don't much care what those strides are (i.e. old vector computers, and maybe GPUs?), one might consider a transpose moving down each off-diagonal in turn, exchanging with the corresponding off-diagonal.

By far the best method is not to transpose at all – make sure that whatever one was going to do next can cope with its input arriving lacking a final transpose.

Note that most routines in the ubiquitous linear algebra package BLAS accept their input matrices in either conventional or transposed form.

There is More Than Multiplication

This lecture has concentrated on the 'trivial' examples of matrix multiplication amd transposes. The idea that different methods need to be used for different problem sizes is much more general, and applies to matrix transposing, solving systems of linear equations, FFTs, etc.

It can make for large, buggy, libraries. For matrix multiplication, the task is valid for multiplying an $n \times m$ matrix by a $m \times p$ matrix. One would hope that any released routine was both correct and fairly optimal for all square matrices, and the common case of one matrix being a vector. However, did the programmer think of testing for the case of multiplying a $1,000,001 \times 3$ matrix by a 3×5 matrix? Probably not. One would hope any released routine was still correct. One might be disappointed by its optimality.

Doing It Oneself

If you are tempted by DIY, it is probably because you are working with a range of problem sizes which is small, and unusual. (Range small, problem probably not small.)

To see if it is worth it, try to estimate the MFLOPS achieved by whatever routine you have readily to hand, and compare it to the processor's peak theoretical performance. This will give you an upper bound on how much faster your code could possibly go. Some processors are notoriously hard to get close to this limit. Note that here the best result for the Core 2 was about 91%, whereas for the Pentium 4 it was only 83%, and for a single core of the Haswell a mere 47%.

If still determined, proceed with a theory text book in one hand, and a stop-watch in the other. And then test the resulting code thoroughly.

Although theory may guide you towards fast algorithms, processors are sufficiently complex and undocumented that the final arbitrator of speed has to be the stopwatch.

Vectorisation and GPUs

Vectorisation, and GPUs

Two topic concerning modern ‘CPU’ design deserve particular mention for being of particular relevance to achieving the best performance on modern processors: vectorisation and GPUs. So this lecture is discusses the hardware aspects of these which are relevant to making code fast and efficient.

What is Vectorisation?

We have seen that CPUs have high throughput when dealing with data-independent operations, as they can then overlap the execution of one operation with another, or even execute them simultaneously if they will execute in different functional units.

Vectorisation uses hardware and software to make simultaneous execution of data-independent operations easier.

Vectorisation and Registers

The first change with vectorisation is that a set of *vector* registers exists, each of which holds multiple elements of the same datatype. In practice 'multiple' always means a power of two, and is generally two, four or eight. (The old vector supercomputers of the later 1980s and early 1990s were more ambitious, and tended to use vector lengths of between 32 and 128.)

Intel's approach is to use general-purpose vector registers of a fixed length in bits. The first ones to support vectors of double-precision floating-point data were those introduced with the 'SSE2' instruction set with the Pentium 4 in 2000. These were 128 bits in size, and so could contain two sixty-four bit floating point doubles, or four thirty-two bit floating point singles. The same registers could also hold integers, as sixteen 8-bit integers, eight sixteen-bit integers, four 32-bit integers, or two 64-bit integers. This ability for the same registers to hold either floating-point or integer data is unusual.

Vector Operations

Instructions for the expected basic arithmetic and logical operations on the vector exist, and they almost all act element-wise, e.g. for multiplication each element of the output vector is the product of the corresponding elements of the two input vectors – not what a mathematician would expect a vector product to do!

Comparison operations exist, and the result they produce has bits set to one where the comparison is true, and zero where it is false.

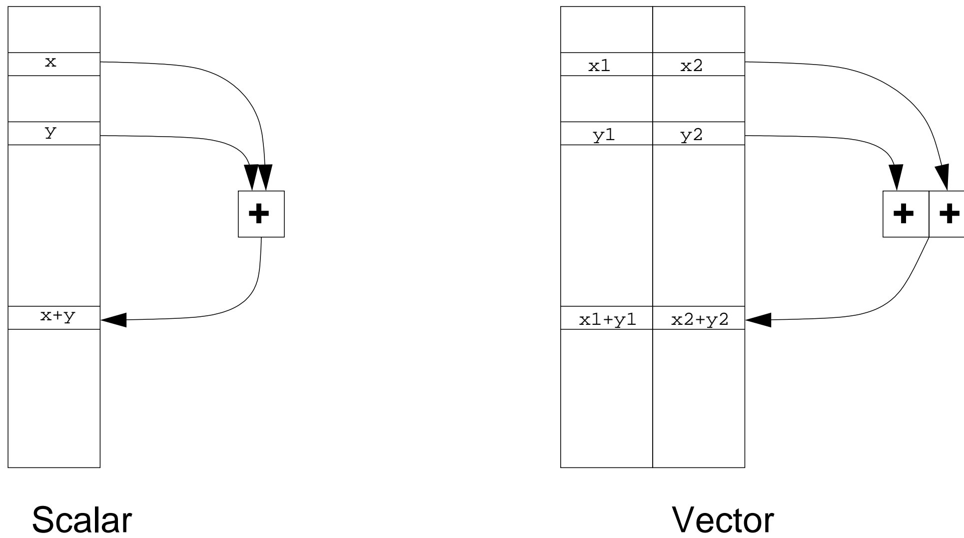
In 2004, Intel introduced instructions which did reductions, such as summing the elements in a vector to produce a scalar.

Vector Functional Units

The Crays of the 1980s and early 1990s, and the Pentium 4, did not really have vector functional units. They had highly pipelined scalar units, and used the fact that an operation on a vector was an operation on a set of independent scalar values, and therefore assuming that the scalar unit could start a new operation every clockcycle, the vector register could feed in a new set of operands each cycle. Indeed, on Cray-like systems one would see instruction timings expressed in ways such as “n+8 cycles” where ‘n’ was the number of elements in the vector in use.

The recent approach from Intel has been rather different. From the Core onwards (2006), most of Intel’s CPUs have been designed with functional units which can operate on a full-width vector simultaneously. In other words, the floating point addition unit on the Core CPU can accept two 64-bit adds, or four 32-bit adds, per clock-cycle.

In pictures



Registers and functional units doubled in size. Instruction fetch and issue logic unchanged. For Intel, the dedicated integer registers and functional units remain scalar, and dedicated vector registers and functional units exist, which can process integer or floating-point data.

But I don’t have vectors!

You sort of do: a scalar is a vector of length one, and there are instructions to say ‘operate on just the bottom element of the vector’. These save no time, except on loads and stores to memory, but do stop junk in the rest of the vector register causing floating point exceptions.

Intel’s long-standing recommendation is that code should never use its old scalar FPU, but only the vector one, with a vector length of one when necessary. Transferring data between the legacy “x87” FPU registers and the new vector registers is tedious, and most compilers never issue “x87” instructions.

Memory costs

Assuming that the elements of a vector are stored consecutively in memory, there can be further advantages. Remember that memory likes consecutive access, and hates random access. Caches store data in lines, which are typically at least 256 bits long, and probably 512 bits.

Loading a single element from memory to cache is equivalent to loading a whole line: caches generally cannot load partial lines. Loading a vector register from a cache line, assuming the internal bus in the CPU is wide enough, might take no longer than loading a scalar into a register.

However, this works best if the vector does not cross cache lines. Correct data alignment can boost performance significantly.

Loads and stores associated with a register of size 2^n bytes should always be from/to addresses which are multiples of 2^n bytes.

Alignment and 2D arrays

In the world of scalars, a compiler could ensure that the start of a double precision array was 8-byte aligned, and then every element of the array would be 8-byte aligned and not cross a cache-line boundary. Now things are harder.

```
real(kind(1d0))::a(1024,1024)
...
do i=1,1024
  a(i,j)=a(i,j)...
enddo
```

```
double a[1024][1024];

for(i=0;i<1024;i++){
  a[j][i]=a[j][i]...
}
```

The loop has unit stride access on `a`, and the first element in the loop is correctly aligned to be loaded with its neighbours as a vector efficiently, whatever the value of `j`.

```
real(kind(1d0))::a(1023,1024)
...
do i=1,1023
  a(i,j)=a(i,j)...
enddo
```

```
double a[1024][1023];

for(i=0;i<1023;i++){
  a[j][i]=a[j][i]...
}
```

Now, if the vector length is four, only one in four values of `j` leads to the first element of the loop being ideally aligned.

A need for speed

Intel has added floating-point vector instructions to its CPUs at least three times: SSE2 (128-bit registers, introduced in 2000, updated with SSE3 in 2004), AVX (256-bit registers, introduced in 2011) and AVX-512 (512-bit registers, introduced 2016 but still absent from most desktops and all laptops in 2018).

An AVX-512 execution unit is a marvellous thing. It can start a fused multiply-add instruction on a vector of eight doubles every clock-cycles. So the one unit can sustain sixteen double precision FP operations per clock cycle, and some of Intel's most recent CPUs have two independent AVX-512 units per core.

The problem comes when such a processor meets SSE2 instructions. It will execute them happily, but, no matter how data independent the SSE2 instructions, only one will start in each AVX-512 unit per cycle. As SSE2 lacks an FMA instruction, this drops the theoretical performance from 16 FLOPS/Hz to 2 FLOPS/Hz per AVX-512 unit.

Increasing the size (or number) of the vector registers always requires OS support, as it changes the amount of state which needs to be saved when switching from one process to another. In theory Intel CPUs will run under old OSes with unsupported features disabled.

A Problem

Does one compile for speed, and sacrifice compatibility with older hardware, or compile for compatibility, and lose most of the performance benefits of modern hardware?

For a major, single-architecture machine such as a local or national supercomputer, clearly one compiles for its architecture: the binary will never be run elsewhere.

Many compilers can produce code which switches execution path at run-time depending on which processor it is executing on. In theory the speed-insensitive parts get compiled to some very general instructions, and then two (or more) versions of the speed-critical parts are produced with run-time selection.

This technique is used by many optimised maths libraries, including OpenBLAS and Intel's MKL. So if the only speed-critical part of one's code is its Lapack calls, one can get this tuning for free. If the maths library is dynamically-linked, even old binaries compiled before a new architecture existed may be able to take advantage of its features.

Trickery

If code compiled using Intel's compiler finds itself running on an AMD processor which supports AVX instructions, will it execute the AVX instruction path, or will it choose to execute the slower SSE2 path, and thus make the AMD processor look less good?

From the documentation of Intel's compiler:

> `-axcode` Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel processors if there is a performance benefit. It also generates a baseline code path. The Intel feature-specific auto-dispatch path is usually more optimized than the baseline path.
>
> The baseline code path is determined by the architecture specified by options `-m` or `-x`. The specified architecture becomes the effective minimum architecture for the baseline code path.
>
> If you specify both the `ax` and `x` options, the baseline code will only execute on Intel processors compatible with the setting specified for the `x`.
>
> If you specify both the `-ax` and `-m` options, the baseline code will execute on non-Intel processors compatible with the setting specified for the `-m` option.
>
> If you specify both the `-ax` and `-march` options, the compiler will not generate Intel-specific instructions.

Subtle Problems

Computing is quite energy-intensive, and modern CPUs are very good at shutting down idle parts of themselves. Current (2017/2018) Xeon Scalable CPUs tend to keep the bottom 128 bits of their vector units active, but if the top part is unused for a few milliseconds, it goes to sleep and then takes a few microseconds to re-awaken. During this period, which is thousands of clock-cycles, 256 bit vector instructions execute by passing twice through the bottom 128 bit part of the functional unit.

The situation with 512 bit instructions is yet worse. Intel CPUs operate over a range of speeds, 'base' to 'turbo', with the higher speeds available when thermal limits permit, and the base speed guaranteed. Only for Xeon Scalable CPUs running AVX-512 instructions, the base speed is far from guaranteed.

Clock speeds

As an example, we consider the Xeon Scalable Gold 6126, a mid-range CPU supporting multiple sockets, and one of which TCM has several.

It has twelve cores and a base clockspeed of 2.6GHz.

Its maximum turbo frequency ranges from 3.3GHz (all cores active) to 3.7GHz (one or two cores active).

When running 256 bit vector instructions, the base frequency drops to 2.2GHz, and the turbo range is 2.9GHz to 3.6GHz.

When running 512 bit vector instructions, the base frequency drops to 1.7GHz, and the turbo range is 2.3GHz to 3.5GHz.

In practice, in a cold machine-room, with the turbo turned off, and all cores active, the clock speed is 2.6GHz unless running 512 bit instructions when it is 2.3GHz.

This means that the theoretical peak performance of this CPU is $12 \times 32 \times 2.3 = 883$ GFLOPS, not $12 \times 32 \times 2.6 = 998$ GFLOPS.

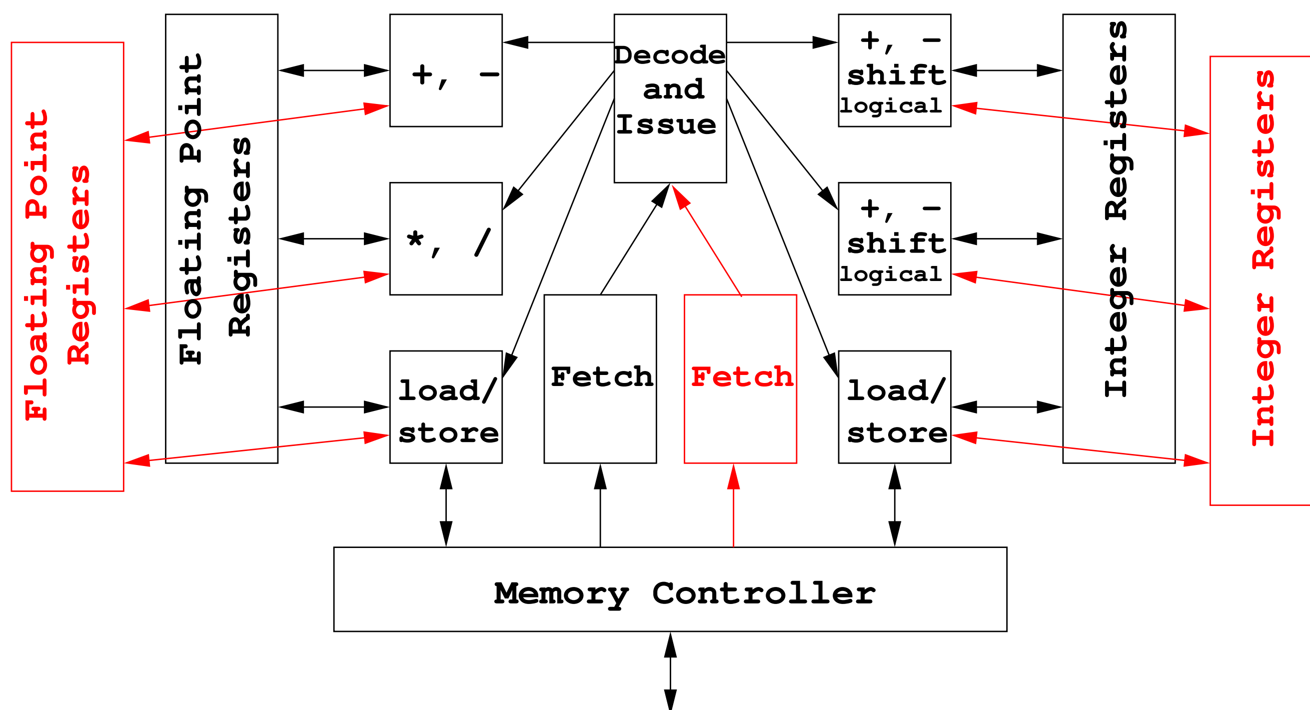
Hyperthreading

The performance of even a scalar CPU is reduced by data dependency problems. There might be capacity in terms of the functional units to execute two instructions simultaneously, or with a large amount of overlap, but, if the second requires as input the result of the first, this cannot be done.

Hyperthreading involves doubling the number of registers on a CPU, and assigning the second set to a new ‘logical’ CPU core. Each ‘logical’ core can run completely different programs, from different users, but sharing the same functional units. However, there will be no data dependencies between these instruction streams, or ‘hyperthreads’, and especially if one hyperthread is stalled waiting for memory, the other might not be, so greater utilisation of the functional units is possible.

From the point of view of the operating system, it appears as though the computer has twice as many cores as are physically present.

Schematic of Typical Hyperthreaded CPU



Hyperthreading's costs

Intel claimed that the extra registers, and extra logic in keeping the separate virtual cores separate, increased the transistor count by about 5%, yet the performance improvement when running multiple processes was typically over 20%.

One area of cost is in the caches, particularly the L1 cache, which is now serving two separate virtual cores. This is likely to lead to a decrease in its hit rate. For normal multitasking, the time to fill the L1 cache, being around 32KB at around 32GB/s, so 1μs, is much less than the time for which a process has exclusive access to a CPU core, which is around 1ms. For most programs having their L1 cache effectively emptied by multitasking every 1ms is less significant than having to share it with another simultaneously-executing hyperthread.

Intel first introduced hyperthreading, with two threads per core, in 2002. It then dropped it with the Core and Core2 in 2006, and re-introduced it with the Nehalem in 2008. Subsequent Intel desktop processors have all had it, save for occasional omissions on the cheapest of any given series.

AMD has been less keen on hyperthreading, and introduced it with its Zen architecture in 2017.

Variations on Hyperthreading

There is no reason to stop at two logic cores (and hence hyperthreads) per real core – some designs have four, or more. Higher numbers of threads are better at hiding memory latencies.

Some variations insist that all instructions issued on the same clock-cycle are from the same hyperthread. Some do not.

There may be differences in whether TLBs, branch predictors, etc. are shared or replicated. The more that is replicated for each logical core, the more the cost in transistors of having hyperthreading.

All make code tuning very hard, as one's run time will be highly dependent on what the second thread is doing.

Hyperthreading's successes

Hyperthreading is excellent at keeping time-sensitive threads which would dislike being suspended for 1ms running. Video decoders may benefit slightly, for instance.

It is good at dealing with disparate processes running simultaneously.

It is less good at dealing with highly synchronised processes. If both threads are competing for the same resource, there is less likely to be a benefit. One obvious example is most dense linear algebra. If the non hyperthreaded code was achieving 90% of the theoretical peak performance of the functional units, there is little room for gain, and considerable potential for loss.

Most HPC centres turn hyperthreading off.

In Linux, if hyperthreading is on, one will see data like the following in `/proc/cpuinfo`. If it is off the two numbers below will be identical.

```
cpu cores       : 4
siblings        : 8
```

GPUs

In many ways the architecture of a GPU shares similar features to that of a CPU with a large vector length. However, it is easier to think about it from a different direction. Whereas one considers a vector CPU as starting as a single scalar CPU, and then having its registers and functional units stretched to form vectors, a better model for a GPU is lots of independent scalar CPU cores which are merged to share the same instruction decoder, and thus all execute in lock-step, each core executing the same instruction of the same program as all other cores, but on different data.

Nvidia's Approach

Nvidia breaks code down into groups of up to 32 threads called a *warp*. A warp executes a single instruction at a time across all threads, and conditional processing is achieved by masking execution on those threads which are not meant to be participating. In other words, instructions for both parts of an if/then/else block are issued, but each individual thread will be masked for one half.

The GPU itself is broken into several 'Streaming Multiprocessors' (SMs), each of which can execute instructions from a small number of warps simultaneously.

A SM

An nvidia SM typically contains a very large number of 32-bit registers, 32,768 or 65,536, and a very large number of rather simple cores which perform 32-bit integer or floating-point operations. These cores are called streaming processors (SPs) or CUDA cores. However, they are more closely equivalent to a standard CPU's functional units: they do not do instruction fetch, decoding or dispatch, nor memory operations.

Separately a SM has some load/store units, and separately again it might have some double-precision floating-point units.

Recent SMs have been partitioned, with each part containing one warp scheduler and its own registers and SPs.

The Pascal GPU (2016) partitions each SM into just two halves. A half of an SM contains 32 SPs, 8 load/store units, 32,768 registers, and 16 double-precision units. The SPs can now perform 16 bit floating point operations as well as 32 bit.

A Pascal

(Figure copyright nvidia)

Hyperthreading?

Why so many registers? 32,768 registers is a lot, even when one considers that they are shared between 32 SPs, and the same registers are used for both integer and floating-point data. Or that double precision values need a pair of registers each.

A single SM is intended to execute multiple warps simulataneously, in a manner similar to hyperthreading, but slightly less transparent. A single thread can access up to 255 registers. A warp, of up to 32 threads, can access up to 8,160. An SM can cope with 64 warps simultaneously, interleaving their instructions, and needs at least four if it is to come close to using all of its registers. If it is executing fewer than two warps, it cannot keep all of its SPs busy.

GPUs and Memory

The main memory on a GPU tends to be small and fast. A Tesla V100 has up to 16GB of local memory, and this has moved on from GDDR5 to HBM2. This has a very wide bus from the GPU, 4096 bits, and runs at a rather modest speed of around 1.75GT/s, for a total bandwidth of around 900GB/s. Earlier generations of GPU tended to use GDDR5 running at up to 7GT/s and with a width of around 384 bits, giving around 350GB/s.

Compared to a standard desktop with a 128 bit memory bus (dual channel DDR), and around 2.5GT/s, thus 40GB/s, this is a lot better. However, higher-end desktops use 256 bit buses (quad channel), and current dual socket machines can have 384 bit buses on each socket, giving a total bandwidth of 240GB/s, not far from that of a GPU.

But today’s (2018) standard desktops can take up to 64GB of memory, and those using quad-channel memory controllers up to 512GB.

GPUs and Economics

In July 2018 I obtained some quotes for a dual socket Xeon Scalable Gold 6130 system with two 16 core 2.1GHz CPUs and 192GB of memory. The quotes were around £8k. The theoretical peak performance figures, assuming that the clock speed drops to 1.9GHz when running AVX512 code, were 1.9TFLOPS and 255GB/s.

Figures for a Tesla V100 are around 7TFLOPS, 900GB/s, and £7,500 from Amazon, but one would need to add around £1,000 for a suitable host computer. Dropping the Xeon system from 192GB to 96GB might save £1,000.

In terms of power the solutions are similar: the peak power usage of the Tesla is 300W (excluding host), and the Xeons are 250W for the pair, excluding the rest of the machine.

So price, similar. Electricity, similar. Memory capacity, Xeon much higher and more expandable. Memory bandwidth, GPU 3½ times better. Double precision performance, GPU 3½ times better. Single precision performance (both twice DP performance) GPU 3½ times better. Half-precision performance, GPU 14 times better.

The hope was that numerical supercomuting would be done on high-end (c £500) GPUs used by gamers. In practice the gamers' GPUs have few double-precision floating point units, leading to different dies being used for supercomputing, and the economies of scale lost, at least in the manufacturing part of the process.

GPUs and Amdahl

It is hard to get the Xeon system up to peak performance. To achieve this, one needs to keep 32 cores busy with vectors of length 8.

The GPU is worse. It has 84 SMs, each needing two warps of length 32 to keep its SPs busy, so one needs 168 warps of length 32 to keep the 5,376 CUDA cores busy (perhaps 168 warps of length 16 for double precision code).

If one ever drops to unvectorised, serial, code, then the CPU core, with a clock-speed about twice that of the GPU, and the ability to issue more instructions per cycle, and to speculate more, is likely to win by a significant margin.

See the hardware part of the MPI course for the reference to Amdahl.

Running where?

It is often the case that parts of a program are suited to running on a GPU, and parts not. This may simply be a way of getting round the small memory capacity of most GPU cards, or a problem with parallelisation.

If one needs to keep transferring data to and from the host's memory and the GPU card, the transfer times can eliminate any potential speed benefit of using the GPU, to the extent that even if the GPU can calculate in zero time, it still loses.

The usual interface to a GPU card is a single PCIe v3 x16 interface, offering about 16GB/s. If an array needs to be copied to and from the GPU card, that is one double per nanosecond. The host CPU can probably execute a few hundred floating point operations per ns, so one needs to ensure that work is sent to the GPU in fairly large chunks.

In particular, there is no point in trying to offload a dot-product to a GPU: on a CPU this is a memory-bound operation, and trying to move the data across a PCIe bus will just make the situation much worse. One needs to move data to a GPU card, and then do as much as possible on it before another data transfer.

GPUs and Programming

Today (2018) Nvidia dominates the hardware market for GPUs used for compute, although AMD GPUs are certainly used (particularly for mining cryptographic currencies). The programming approach which dominates on Nvidia hardware is CUDA.

CUDA is not ideal. It is unsupported by Intel's compiler, which is generally the most popular compiler for HPC. The current compilers support only Nvidia's GPUs.

Alternatives are generally worse. OpenCL is more vendor-neutral (it runs on both Nvidia and AMD GPUs), but it is considered harder to program, and though once an integral part of MacOS, Apple now deprecates it.

Directive-based approaches exist (OpenACC, and OpenMP since version 4.0 (2013)). Again vendor-neutral, but they rarely get close to the performance of CUDA.

Finally there are libraries which run on GPUs. But these tend to restrict one to the model of move data to GPU, perform operation, move back. This can lead to a lot of unnecessary, and slow, data movement.

GPUs: the future?

Predicting the future in IT is never wise, but one slide of foolishness will not hurt much.

GPUs may gain access to the host's main memory equal to that of the CPU, rather than being the wrong side of a PCIe bus. That might solve the data transfer problem, although it might also eliminate their memory bandwidth advantage, as currently GPUs enjoy faster memory than their hosts.

If one's interest is in integer or single-precision data types of the sort needed for 3D graphics/games, then very attractive hardware will be available for around £500. If one is firmly biased towards double-precision work, then the performance of the mass-market hardware is likely to be poor, and the specialist HPC hardware likely to remain expensive.

OpenACC may well be more completely absorbed into and superceded by OpenMP.

Memory Management

Memory: a Programmer's Perspective

From a programmer's perspective memory is simply a linear array into which bytes are stored. The array is indexed by a pointer which runs from 0 to 2^{32} (4GB) on 32 bit machines, or 2^{64} (16EB) on 64 bit machines. This section assumes 32 bit addresses in most cases, not because 32 bit machines are relevant to scientific computing today, but because the principles are the same as for 64 bit machines, and 64 bit addresses are rather long for use as examples.

The memory has no idea what type of data it stores: integer, floating point, program code, text, it's all just bytes.

An address may have one of several attributes:

Invalid	not allocated
Read only	for constants and program code
Executable	for program code, not data
Shared	for inter-process communication
On disk	paged to disk to free up real RAM

(Valid virtual addresses on current 64 bit machines reach only 2^{48} (256TB). So far no-one is complaining. To go further would complicate the page table (see below).)

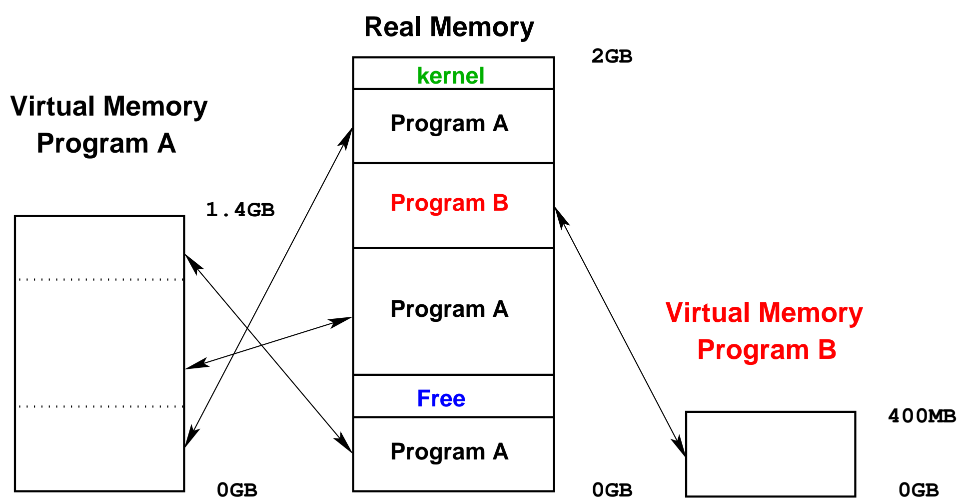
Pages

In practice memory is broken into *pages*, contiguous regions, often of 4KB, which are described by just a single set of the above attributes. When the operating system allocates memory to a program, the allocation must be an integer number of pages. If this results in some extra space, `malloc()` or `allocate()` will notice, and may use that space in a future allocation without troubling the operating system.

Modern programs, especially those written in C or, worse, C++, do a lot of allocating and deallocating of small amounts of memory. Some remarkably efficient procedures have been developed for dealing with this. Ancient programs, such as those written in Fortran 77, do no run-time allocation of memory. All memory is fixed when the program starts.

Pages also allow for a mapping to exist between *virtual* addresses as seen by a process, and *physical* addresses in hardware.

No Fragmentation



Pages also have an associated location in real, physical memory. In this example, program A believes that it has an address space extending from 0MB to 1400MB, and program B believes it has a distinct space extending from 0MB to 400MB. Neither is aware of the mapping of its own virtual address space into physical memory, or whether that mapping is contiguous.

Splendid Isolation

This scheme gives many levels of isolation.

Each process is able to have a contiguous address space, starting at zero, regardless of what other processes are doing.

No process can accidentally access another process's memory, for no process is able to use physical addresses. They have to use virtual addresses, and the operating system will not allow two virtual addresses to map to the same physical address (except when this is really wanted).

If a process attempts to access a virtual address which it has not been granted by the operating system, no mapping to a physical address will exist, and the access must fail. A segmentation fault.

A virtual address is unique only when combined with a process ID (deliberate sharing excepted).

Fast, and Slow

This scheme might appear to be very slow. Every memory access involves a translation from a virtual address to a physical address. Large translation tables (page tables) are stored in memory to assist. These are stored at known locations in physical memory, and the kernel, unlike user processes, can access physical memory directly to avoid a nasty catch-22.

Every CPU has a cache dedicated to storing the results of recently-used page table look-ups, called the TLB. This eliminates most of the speed penalty, except for random memory access patterns.

A TLB is so essential for performance with virtual addressing that the 80386, the first Intel processor to support virtual addressing, had a small (32 entry) TLB, but no other cache.

Page Tables

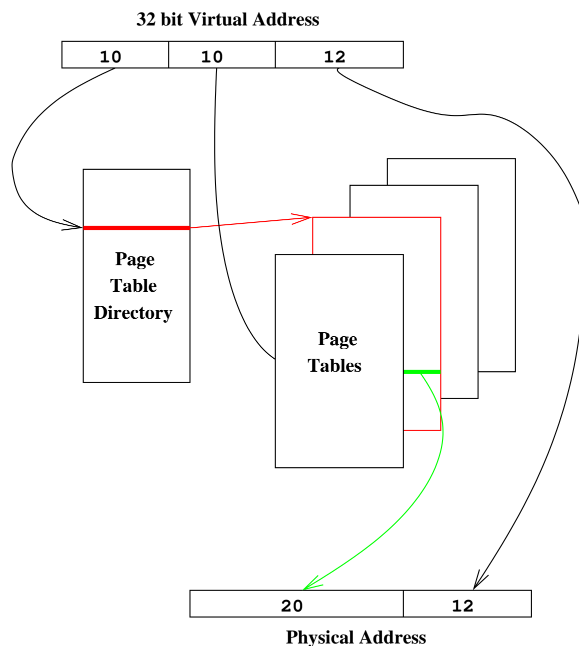
A 32 bit machine will need a four-byte entry in a page table per page. With 4KB pages, this could be done with a 4MB page table per process covering the whole of its virtual address space. However, for processes which make modest use of virtual address space, this would be rather inefficient. It would also be horrific in a 64 (or even 48) bit world.

So the page table is split into two. The top level describes blocks of 1024 pages (4MB). If no address in that range is valid, the top level table simply records this invalidity. If any address is valid, the top level table then points to a second level page table which contains the 1024 entries for that 4MB region. Some of those entries may be invalid, and some valid.

The logic is simple. For a 32 bit address, the top ten bits index the top level page table, the next ten index the second level page table, and the final 12 an address within the 4KB page pointed to by the second level page table.

(This sort of zero effort decoding also applies to cache lines, tags, and offsets within lines.)

Page Tables in Action



For a 64 bit machine, page table entries must be eight bytes. So a 4KB page contains just 512 (2^9) entries. Intel currently uses a four level page table for '64 bit' addressing, giving $4 \times 9 + 12 = 48$ bits. The Alpha processor used a three level table and an 8KB page size, giving $3 \times 10 + 13 = 43$ bits.

Efficiency

This is still quite a disaster. Every memory reference now requires two or three additional accesses to perform the virtual to physical address translation.

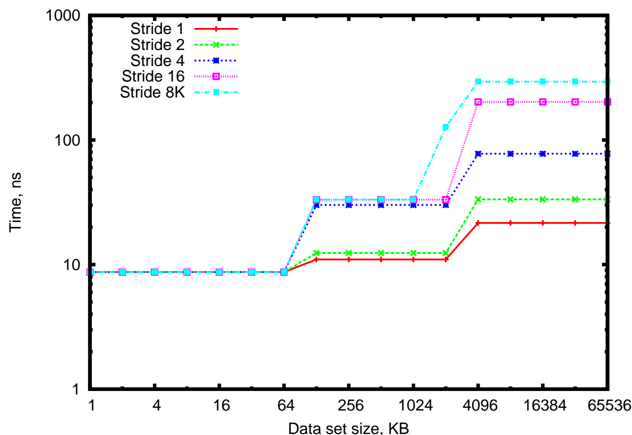
Fortunately, the CPU understands pages sufficiently well that it remembers where to find frequently-referenced pages using a special cache called a TLB. This means that it does not have to keep asking the operating system where a page has been placed.

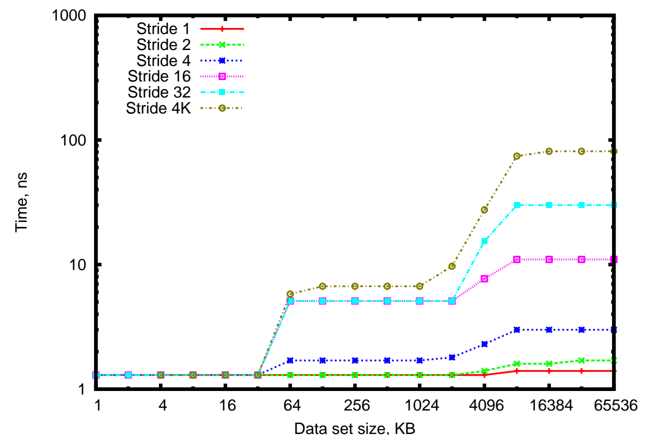
Just like any other cache, TLBs vary in size and associativity, and separate instruction and data TLBs may be used. A TLB rarely contains more than 1024 entries, often far fewer.

Even when a TLB miss occurs, it is rarely necessary to fetch a page table from main memory, as the relevant tables are usually still in secondary cache, left there by a previous miss.

TLB = translation lookaside buffer
ITLB = instruction TLB, DTLB = data TLB if these are separate

TLBs at work





The left is a repeat of the graph on page 81, but with an 8KB stride added. The XP900 uses 8KB pages, and has a 128 entry DTLB. Once the data set is over 1MB, the TLB is too small to hold its pages, and, with an 8KB stride, a TLB miss occurs on every access, adding 92ns.

The right is a repeat of the Core 2 graph from page 82, with a 4KB stride added. The Core 2 uses 4KB pages, and has a 256 entry DTLB. Some more complex interactions are occuring here, but it finishes up with a 50ns penalty.

Given that three levels of page table must be accessed, it is clear that most of the relevant parts of the page table were in cache. So the 92ns and 50ns recovery times for a TLB miss are best cases – with larger data sets it can get worse. The Alpha is losing merely 43 clock cycles, the Core 2 about 120. As the data set gets yet larger, TLB misses will be to page tables not in cache, and random access to a 2GB array results in a memory latency of over 150ns on the Core 2.

More paging

Having suffering one level of translation from virtual to physical addresses, it is conceptually easy to extend the scheme slightly further. Suppose that the OS, when asked to find a page, can go away, read it in from disk to physical memory, and then tell the CPU where it has put it. This is what all modern OSes do (UNIX, OS/2, Win9x / NT, MacOS X), and it merely involves putting a little extra information in the page table entry for that page.

If a piece of real memory has not been accessed recently, and memory is in demand, that piece will be paged out to disk, and reclaimed automatically (if slowly) if it is needed again. Such a reclaiming is also called a page fault, although in this case it is not fatal to the program.

Rescuing a page from disk will take about 10ms, compared with under 100ns for hitting main memory. If just one in 10^5 memory accesses involve a page-in, the code will run at half speed, and the disk will be audibly ‘thrashing’.

The union of physical memory and the page area on disk is called *virtual memory*. Virtual addressing is a prerequisite for virtual memory, but the terms are not identical.

Less paging

Certain pages should not be paged to disk. The page tables themselves are an obvious example, as is much of the kernel and parts of the disk cache.

Most OSes (including UNIX) have a concept of a *locked*, that is, unpageable, page. Clearly all the locked pages must fit into physical memory, so they are considered to be a scarce resource. On UNIX only the kernel or a process running with root privilege can cause its pages to be locked.

Much I/O requires locked pages too. If a network card or disk drive wishes to write some data into memory, it is too dumb to care about virtual addressing, and will write straight to a physical address. With locked pages such pages are easily reserved.

Certain ‘real time’ programs which do not want the long delays associated with recovering pages from disk request that their pages are locked. Examples include CD/DVD writing software, or video players.

Blatant Lies

Paging to disk as above enables a computer to pretend that it has more RAM than it really does. This trick can be taken one stage further. Many OSes are quite happy to allocate virtual address space, leaving a page table entry which says that the address is valid, not yet ever been used, and has no physical storage associated with it. Physical storage will be allocated on first use. This means that a program will happily pass all its `malloc()` / `allocate` statements, and only run into trouble when it starts trying to use the memory.

The `ps` command reports both the virtual and physical memory used:

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS TTY     STAT START  TIME COMMAND
spqr1 20241  100 12.7 711764 515656 pts/9  Rl+  13:36  3:47 castep si64
```

RSS – Resident Set Size (i.e. physical memory use). Will be less than the physical memory in the machine. `%MEM` is the ratio of this to the physical memory of the machine, and thus can never exceed 100.

VSZ – Virtual SiZe, i.e. total virtual address space allocated. Cannot be smaller than RSS.

The Problem with Lying

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS     TTY  STAT START  TIME COMMAND
spqr1 25175 98.7 25.9 4207744 1049228 pts/3 R+   14:02  0:15 ./a.out
```

Currently this is fine – the process is using just under 26% of the memory. However, the `VSZ` field suggests that it has been promised 104% of the physical memory. This could be awkward.

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS     TTY  STAT START  TIME COMMAND
spqr1 25175 39.0 90.3 4207744 3658252 pts/0 D+   14:02  0:25 ./a.out
```

Awkward. Although the process does no I/O its status is ‘D’ (waiting for ‘disk’), its share of CPU time has dropped (though no other process is active), and inactive processes have been badly squeezed. At this point Firefox had an RSS of under 2MB and was extremely slow to respond. It had over 50MB before it was squeezed.

Interactive users will now be very unhappy, and if the computer had another GB that program would run almost three times faster.

One can experiment with `ulimit -v` to limit a process’s virtual address space.

Grey Areas – How Big is Too Big?

It is hard to say precisely. If a program allocates one huge array, and then jumps randomly all over it, then the entirety of that array must fit into physical memory, or there will be a huge penalty. If a program allocates two large arrays, spends several hours with the first, then moves it attention to the second, the penalty if only one fits into physical memory at a time is slight. Total usage of physical memory is reported by `free` under Linux. Precise interpretation of the fields is still hard.

```
$ free
             total      used      free    shared   buffers    cached
Mem:       4050700    411744   3638956         0      8348    142724
-/+ buffers/cache:    260672   3790028
Swap:      6072564     52980   6019584
```

The above is fine. The below isn't. Don't wait for `free` to hit zero – it won't.

```
$ free
             total      used      free    shared   buffers    cached
Mem:       4050700   4021984     28716         0       184    145536
-/+ buffers/cache:   3876264    174436
Swap:      6072564    509192   5563372
```


Page sizes

A page is the smallest unit of memory allocation from OS to process, and the smallest unit which can be paged to disk. Large page sizes result in wasted memory from allocations being rounded up, longer disk page in and out times, and a coarser granularity on which unused areas of memory can be detected and paged out to disk. Small page sizes lead to more TLB misses, as the virtual address space 'covered' by the TLB is the number of TLB entries multiplied by the page size.

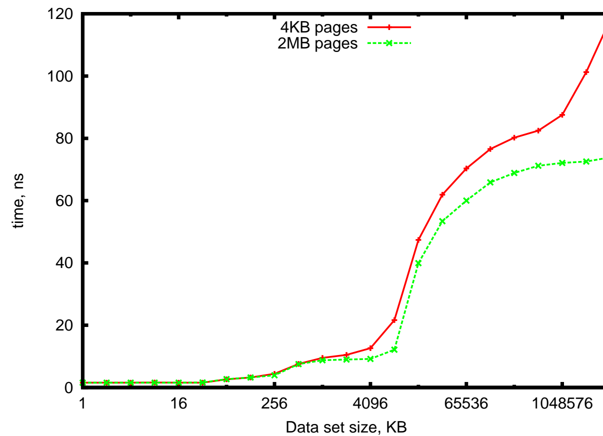
Large-scale scientific codes which allocate hundreds of MB of memory benefit from much larger page sizes than a mere 4KB. However, a typical UNIX system has several dozen small processes running on it which would not benefit from a page size of a few MB.

Intel's processors do support 2MB pages, but support in Linux is unimpressive prior to 2.6.38. Support from Solaris for the page sizes offered by the (ancient) UltraSPARC III (8K, 64K, 512K and 4MB) is much better.

DEC's Alpha solves this issue in another fashion, by allowing one TLB entry to refer to one, eight, 64 or 512 consecutive pages, thus effectively increasing the page size.

Large Pages in Linux

From kernel 2.6.38, Linux will use large pages (2MB) by default when it can. This reduces TLB misses when jumping randomly over large arrays.



The disadvantage is that sometimes fragmentation in physical memory will prevent Linux from using (as many) large pages. This will make code run slower, and the poor programmer will have no idea what has happened.

This graph can be compared with that on page 161, noting that here a random access pattern is used, the y axis is not logarithmic, the processor is an Intel Sandy Bridge, and the x axis is extended another factor of 64.

Expectations

The Sandy Bridge CPU used to generate that graph has a 32KB L1 cache, a 256KB L2, and a 8MB L3. If one assumes that the access times are 1.55ns, 3.9ns, 9.5ns for those, and for main memory 72.5ns, then the line for 2MB pages can be reproduced remarkably accurately. (E.g. at 32MB assume one quarter of accesses are lucky and are cached in L3 (9.5ns), the rest are main memory (72.5ns), so expect 56.7ns. Measured 53.4ns.)

With 4KB pages, the latency starts to increase again beyond about 512MB. The cause is the last level of the page table being increasingly likely to have been evicted from the last level of cache by the random access on the data array. If the TLB miss requires a reference to a part of the page table in main memory, it must take at least 72ns. This is probably happening about half of the time for the final data point (4GB).

This graph shows very clearly that ‘toy’ computers hate big problems: accessing large datasets can be *much* slower than accessing smaller ones, although the future is looking (slightly) brighter.

Caches and Virtual Addresses

Suppose we have a two-way associative 2MB cache. This means that we can cache any contiguous 2MB region of physical memory, and any two physical addresses which are identical in their last 20 bits.

Programs works on virtual addresses. The mapping from virtual to physical preserves the last 12 bits (assuming 4KB pages), but is otherwise unpredictable. A 2MB region of virtual address space will be completely cacheable only for some mappings. If one is really unlucky, a mere 12KB region of virtual address space will map to three physical pages whose last 20 bits are all identical. Then this cannot be cached. A random virtual to physical mapping would make caching all of a 2MB region very unlikely.

Most OSes do magic (page colouring) which reduces, or eliminates, this problem, but Linux does not. This is particularly important if a CPU's L1 cache is larger than its associativity multiplied by the OS's page size (AMD Athlon / Opteron, but not Intel). When the problem is not eliminated, one sees variations in runtimes as a program is run repeatedly (and the virtual to physical mapping changes), and the expected sharp steps in performance as arrays grow larger than caches are slurred.

Segments

A program uses memory for many different things. For instance:

- The code itself
- Shared libraries
- Statically allocated uninitialised data
- Statically allocated initialised data
- Dynamically allocated data
- Temporary storage of arguments to function calls and of local variables

These areas have different requirements.

Segments

Text

Executable program code, including code from statically-linked libraries. Sometimes constant data ends up here, for this segment is read-only.

Data

Initialised data (numeric and string), from program and statically-linked libraries.

BSS

Uninitialised data of fixed size. Unlike the data segment, this will not form part of the executable file. Unlike the heap, the segment is of fixed size.

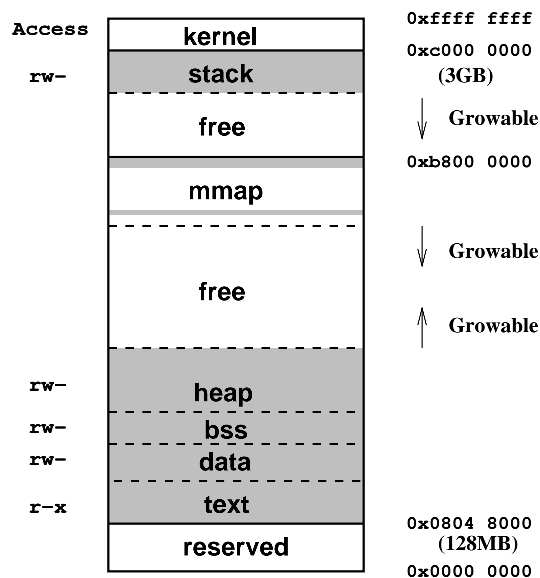
heap

Area from which `malloc()` / `allocate()` traditionally gain memory.

stack

Area for local temporary variables in (recursive) functions, function return addresses, and arguments passed to functions.

A Linux Memory Map



This is roughly the layout used by Linux 2.6 on 32 bit machines, and *not to scale*.

The `mmap` region deals with shared libraries and large objects allocated via `malloc`, whereas smaller `malloc`ed objects are placed on the heap in the usual fashion. Earlier versions grew the `mmap` region *upwards* from about 1GB (0x4000 0000).

Note the area around zero is reserved. This is so that null pointer dereferencing will fail: ask a C programmer why this is important.

What Went Where?

Determining to which of the above data segments a piece of data has been assigned can be difficult. One would strongly expect C's `malloc` and F90's `allocate` to reserve space on the heap. Likewise small local variables tend to end up on the stack.

Large local variables really ought not go on the stack: it is optimised for the low-overhead allocation and deletion needed for dealing with lots of small things, but performs badly when a large object lands on it. However compilers sometimes get it wrong.

UNIX limits the size of the stack segment and the heap, which it 'helpfully' calls 'data' at this point. See the '`ulimit`' command (`[ba]sh`).

Some UNIXes can also limit the total virtual address space used (`ulimit -v`). Some claim to be able to limit the resident set size (`ulimit -m`), but few actually do (Linux does not).

Because `ulimit` is an internal shell command, it is documented in the shell man pages (e.g. '`man bash`'), and does not have its own man page.

Sharing

If multiple copies of the same program or library are required in memory, it would be wasteful to store multiple identical copies of their unmodifiable read-only pages. Hence many OSes, including UNIX, keep just one copy in memory, and have many virtual addresses refering to the same physical address. A count is kept, to avoid freeing the physical memory until no process is using it any more!

UNIX does this for shared libraries and for executables. Thus the memory required to run three copies of Firefox is less than three times the memory required to run one, even if the three are being run by different users. It also greatly reduces program start-up times if their shared libraries are already in memory and being used by another process.

Two programs are considered identical by UNIX if they are on the same device and have the same inode. See elsewhere for a definition of an inode.

If an area of memory is shared, the `ps` command apportions it appropriately when reporting the RSS size. If the whole libc is being shared by ten processes, each gets merely 10% accounted to it.

`mmap`

It has been shown that the OS can move data from physical memory to disk, and transparently move it back as needed. However, there is also an interface for doing this explicitly. The `mmap` system call requests that the kernel set up some page tables so that a region of virtual address space is mapped onto a particular file. Thereafter reads and writes to that area of 'memory' actually go through to the underlying file.

The reason this is of interest, even to Fortran programmers, is that it is how all executable files and shared libraries are loaded. It is also how large dynamic objects, such as the result of large `allocate` / `malloc` calls, get allocated. They get a special form of `mmap` which has no physical file associated with it.

Heap vs `mmap`

Consider the following code:

```
a=malloc(1024*1024*1024); b=malloc(1); free(a)
```

(in the real world one assumes that something else would occur before the final `free`).

With a single heap, the heap now has 1GB of free space, followed by a single byte which is in use. Because the heap is a single contiguous object with just one moveable end, there is no way of telling the OS that is can reclaim the unused 1GB. That memory will remain with the program and be available for its future allocations. The OS does not know that its current contents are no longer required, so its contents must be preserved, either in physical memory or in a page file. If the program (erroneously) tries accessing that freed area, it will succeed.

Had the larger request resulted in a separate object via `mmap`, then the `free` would have told the kernel to discard the memory, and to ensure that any future erroneous accesses to it result in segfaults.

Automatically done

Currently by default objects larger than 128KB allocated via `malloc` are allocated using `mmap`, rather than via the heap. The size of allocation resulting will be rounded up to the next multiple of the page size (4KB). Most Fortran runtime libraries end up calling `malloc` in response to `allocate`. A few do their own heap management, and only call `brk`, which is the basic call to change the size of the heap with no concept of separate objects existing within the heap.

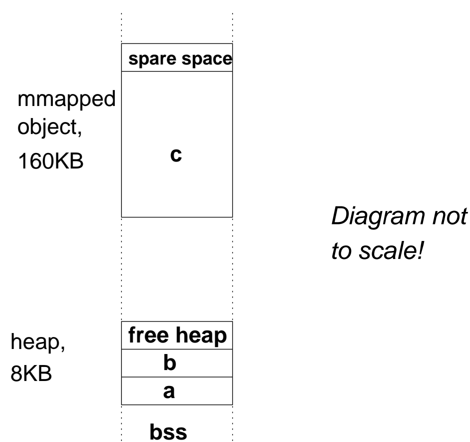
Fortran 90 has an unpleasant habit of placing large temporary and local objects on the stack. This can cause problems, and can be tuned with options such as `-heap-arrays` (ifort) and `-static-data` (Open64).

Objects allocated via `mmap` get placed in a region which lies between the heap and the stack. On 32 bit machines this can lead to the heap (or stack) colliding with this region. On 64 bit machines, there is no problem.

Heap layout

```
double precision, allocatable :: a(:),b(:),c(:)
allocate (a(300),b(300),c(20000))
```

In the absence of other allocations, one would expect the heap to contain `a` followed by `b`. This is 600 doubles, 4,800 bytes, so the heap will be rounded to 8KB (1024 doubles), the next multiple of 4KB. The array `c`, being over 128KB, will go into a separate object via `mmap`, and this will be 160KB, holding 20,480 doubles.



More segfaults

So attempts to access elements of `c` between one and 20,480 will work, and for `a` indices between one and 300 will find `a`, between 301 and 600 will find `b`, and 601 and 1024 will find free space. Only `a(1025)` will cause a segfault. For indices less than one, `c(0)` would be expected to fail, but `b(-100)` would succeed, and probably hit `a(200)`. And `a(-100)` is probably somewhere in the static data section preceeding the heap, and fine.

Array overwriting can go on for a long while before segfaults occur, unless a pointer gets overwritten, and then dereferenced, in which case the resulting address is usually invalid, particularly in a 64 bit world where the proportion of 64 bit numbers which are valid addresses is low.

Fortran compilers almost always support a `-C` option for checking array bounds. It very significantly slows down array accesses – use it for debugging, not real work! The `-g` option increases the chance that line numbers get reported, but compilers differ in how much information does get reported.

C programmers using `malloc()` are harder to help. But they may wish to ask Google about Electric Fence.

Theory in Practice

```
$ cat test.f90
double precision, allocatable :: a(:),b(:),c(:)

allocate (a(300),b(300),c(20000))
a=0
b(-100)=5

write(*,*)'Maximum value in a is ',maxval(a), &
          ' at location ',maxloc(a)
end

$ ifort test.f90 ; ./a.out
 Maximum value in a is  5.00000000000000  at location  202
$ f95 test.f90 ; ./a.out
 Maximum value in a is  5.0  at location  204
$ gfortran test.f90 ; ./a.out
 Maximum value in a is  5.0000000000000000  at location  202
$ openf90 test.f90 ; ./a.out
 Maximum value in a is  0.E+0  at location  1
```

−C

```
$ ifort -C -g test.f90 ; ./a.out
forrtl: severe (408): fort: (3): Subscript #1 of the array B
has value -100 which is less than the lower bound of 1

$ f95 -C -g test.f90 ; ./a.out
 ******  FORTRAN RUN-TIME SYSTEM  ******
Subscript out of range. Location:  line 5 column 3 of 'test.f90'
Subscript number 1 has value -100 in array 'B'
Aborted

$ gfortran -C -g test.f90 ; ./a.out
 Maximum value in a is  5.0000000000000000   at location   202
$ gfortran -fcheck=bounds -g test.f90 ; ./a.out
At line 5 of file test.f90
Fortran runtime error: Index '-100' of dimension 1 of array 'b'
below lower bound of 1

$ openf90 -C -g test.f90 ; ./a.out
lib-4964 : WARNING
  Subscript is out of range for dimension 1 for array
  'B' at line 5 in file 'test.f90',
  diagnosed in routine '__f90_bounds_check'.
 Maximum value in a is  0.E+0  at location  1
```


Disclaimer

By the time you see this, it is unlikely that any of the above examples is with the current version of the compiler used. These examples are intended to demonstrate that different compilers are different. That is why I have quite a collection of them!

`ifort`: Intel's compiler, v 11.1
`f95`: Sun's compiler, Solaris Studio 12.2
`gfortran`: Gnu's compiler, v 4.5
`openf90`: Open64 compiler, v 4.2.4

Four compilers. Only two managed to report line number, and which array bound was exceeded, and the value of the errant index.

The Stack Layout

Address	Contents	Frame Owner
	...	calling
	2nd argument	function
`%ebp+8`	1st argument	
`%ebp+4`	return address	
`%ebp`	previous `%ebp`	
	local	current
	variables	function
	etc.	
`%esp`	end of stack	

The stack grows downwards, and is divided into frames, each frame belonging to a function which is part of the current call tree. Two registers are devoted to keeping it in order.

Memory Maps in Action

Under Linux, one simply needs to examine `/proc/[pid]/maps` using `less` to see a snapshot of the memory map for any process one owns. It also clearly lists shared libraries in use, and some of the open files. Unfortunately it lists things upside-down compared to our pictures above.

The example on the next page clearly shows a program with the bottom four segments being text, data, bss and heap, of which text and bss are read-only. In this case mmaped objects are growing downwards from `f776 c000`, starting with shared libraries, and then including large malloced objects.

The example was from a 32 bit program running on 64 bit hardware and OS. In this case the kernel does not need to reserve such a large amount of space for itself, hence the stack is able to start at `0xfffb 9000` not `0xc000 0000`, and the start of the `mmap` region also moves up by almost 1GB.

Files in `/proc` are not real files, in that they are not physically present on any disk drive. Rather attempts to read from these 'files' are interpretted by the OS as requests for information about processes or other aspects of the system.

The machine used here does not set read and execute attributes separately – any readable page is executable.

The Small Print

```
$ tac /proc/20777/maps
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
fff6e000-fffb9000 rwxp 00000000 00:00 0          [stack]
f776b000-f776c000 rwxp 0001f000 08:01 435109     /lib/ld-2.11.2.so
f776a000-f776b000 r-xp 0001e000 08:01 435109     /lib/ld-2.11.2.so
f7769000-f776a000 rwxp 00000000 00:00 0
f774b000-f7769000 r-xp 00000000 08:01 435109     /lib/ld-2.11.2.so
f7744000-f774b000 rwxp 00000000 00:00 0
f773e000-f7744000 rwxp 00075000 00:13 26596314   /opt/intel/11.1-059/lib/ia32/libguide.so
f76c8000-f773e000 r-xp 00000000 00:13 26596314   /opt/intel/11.1-059/lib/ia32/libguide.so
f76a7000-f76a9000 rwxp 00000000 00:00 0
f76a6000-f76a7000 rwxp 00017000 08:01 435034     /lib/libpthread-2.11.2.so
f76a5000-f76a6000 r-xp 00016000 08:01 435034     /lib/libpthread-2.11.2.so
f768e000-f76a5000 r-xp 00000000 08:01 435034     /lib/libpthread-2.11.2.so
f768d000-f768e000 rwxp 00028000 08:01 435136     /lib/libm-2.11.2.so
f768c000-f768d000 r-xp 00027000 08:01 435136     /lib/libm-2.11.2.so
f7664000-f768c000 r-xp 00000000 08:01 435136     /lib/libm-2.11.2.so
f7661000-f7664000 rwxp 00000000 00:00 0
f7660000-f7661000 rwxp 00166000 08:01 435035     /lib/libc-2.11.2.so
f765e000-f7660000 r-xp 00164000 08:01 435035     /lib/libc-2.11.2.so
f765d000-f765e000 ---p 00164000 08:01 435035     /lib/libc-2.11.2.so
f74f9000-f765d000 r-xp 00000000 08:01 435035     /lib/libc-2.11.2.so
f74d4000-f74d5000 rwxp 00000000 00:00 0
f6fac000-f728a000 rwxp 00000000 00:00 0
f6cec000-f6df4000 rwxp 00000000 00:00 0
f6c6b000-f6c7b000 rwxp 00000000 00:00 0
f6c6a000-f6c6b000 ---p 00000000 00:00 0
f6913000-f6b13000 rwxp 00000000 00:00 0
f6912000-f6913000 ---p 00000000 00:00 0
f6775000-f6912000 rwxp 00000000 00:00 0
097ea000-0ab03000 rwxp 00000000 00:00 0          [heap]
0975c000-097ea000 rwxp 01713000 08:06 9319119    /scratch/castep
0975b000-0975c000 r-xp 01712000 08:06 9319119    /scratch/castep
08048000-0975b000 r-xp 00000000 08:06 9319119    /scratch/castep
```


The Madness of C

```
#include<stdio.h>
#include<stdlib.h>

void foo(int *a, int *b);

int main(void){
  int *a,*b;

  a=malloc(sizeof(int));
  b=malloc(sizeof(int));

  *a=2;*b=3;

  printf("The function main starts at address  %.8p\n",main);
  printf("The function foo  starts at address  %.8p\n",foo);

  printf("Before call:\n\n");
  printf("a is a pointer. It is stored at address %.8p\n",&a);
  printf("                It points to address    %.8p\n",a);
  printf("                It points to the value  %d\n",*a);
  printf("b is a pointer. It is stored at address %.8p\n",&b);
  printf("                It points to address    %.8p\n",b);
  printf("                It points to the value  %d\n",*b);

  foo(a,b);

  printf("\nAfter call:\n\n");
  printf("                 a points to the value  %d\n",*a);
```

```
  printf("                  b points to the value  %d\n",*b);

  return 0;
}

void foo(int *c, int *d){

  printf("\nIn function:\n\n");

  printf("Our return address is                   %.8p\n\n",*(&c-1));

  printf("c is a pointer. It is stored at address %.8p\n",&c);
  printf("                It points to address    %.8p\n",c);
  printf("                It points to the value  %d\n",*c);
  printf("d is a pointer. It is stored at address %.8p\n",&d);
  printf("                It points to address    %.8p\n",d);
  printf("                It points to the value  %d\n",*d);

  *c=5;
  *(*(&c+1))=6;
}
```


The Results of Madness

```
The function main starts at address  0x08048484
The function foo  starts at address  0x080485ce
Before call:

a is a pointer. It is stored at address 0xbfdf8dac
                It points to address    0x0804b008
                It points to the value  2
b is a pointer. It is stored at address 0xbfdf8da8
                It points to address    0x0804b018
                It points to the value  3

In function:

Our return address is                   0x0804858d

c is a pointer. It is stored at address 0xbfdf8d90
                It points to address    0x0804b008
                It points to the value  2
d is a pointer. It is stored at address 0xbfdf8d94
                It points to address    0x0804b018
                It points to the value  3

After call:

                 a points to the value  5
                 b points to the value  6
```

The Explanation

```
0xbfdf ffff   approximate start of stack
   ....
0xbfbf 8da8   local variables in main()
   ....
0xbfdf 8d94   second argument to function foo()
0xbfdf 8d90   first argument
0xbfdf 8d8c   return address
   ....
0x0fdf 8d??   end of stack

0x0804 b020   end of heap
0x0804 b018   the value of b is stored here
0x0804 b008   the value of a is stored here
0x0804 b000   start of heap

0x0804 85ce   start of foo() in text segment
0x0804 858d   point at which main() calls foo()
0x0804 8484   start of main() in text segment
```

And if you note nothing else, note that the function foo managed to manipulate its second argument using merely its first argument.

(This example assumes a 32-bit world for simplicity.)

Compilers & Optimisation

Optimisation

Optimisation is the process of producing a machine code representation of a program which will run as fast as possible. It is a job shared by the compiler and programmer.

The compiler uses the sort of highly artificial intelligence that programs have. This involves following simple rules without getting bored halfway through.

The human will be bored before he starts to program, and will never have followed a rule in his life. However, it is he who has the Creative Spirit.

This section discussed some of the techniques and terminology used.

Loops

Loops are the only things worth optimising. A code sequence which is executed just once will not take as long to run as it took to write. A loop, which may be executed many, many millions of times, is rather different.

```
do i=1,n
  x(i)=2*pi*i/k1
  y(i)=2*pi*i/k2
enddo
```

Is the simple example we will consider first, and Fortran will be used to demonstrate the sort of transforms the compiler will make during the translation to machine code.

Simple and automatic

CSE

```
do i=1,n
  t1=2*pi*i
  x(i)=t1/k1
  y(i)=t1/k2
enddo
```

Common Subexpression Elimination. Rely on the compiler to do this.

Invariant removal

```
t2=2*pi
do i=1,n
  t1=t2*i
  x(i)=t1/k1
  y(i)=t1/k2
enddo
```

Rely on the compiler to do this.

Division to multiplication

```
t2=2*pi
t3=1/k1              t1=2*pi/k1       t1=2*pi/k1
t4=1/k2              t2=2*pi/k2       t2=2*pi/k2
do i=1,n             do i=1,n         do i=1,n
  t1=t2*i                               t=real(i,kind(1d0))
  x(i)=t1*t3           x(i)=i*t1        x(i)=t*t1
  y(i)=t1*t4           y(i)=i*t2        y(i)=t*t2
enddo                enddo            enddo
```

From left to right, increasingly optimised versions of the loop after the elimination of the division.

The compiler shouldn't default to this, as it breaks the IEEE standard subtly. However, there will be a compiler flag to make this happen: find it and use it!

Conversion of `x**2` to `x*x` will be automatic.

Remember multiplication is many times faster than division, and many many times faster than logs and exponentiation.

Some compilers now do this by default, defaulting to breaking IEEE standards for arithmetic. I prefered the more Conservative world in which I spent my youth.

Another example

```
  y=0
  do i=1,n
    y=y+x(i)*x(i)
  enddo
```

As machine code has no real concept of a loop, this will need converting to a form such as

```
  y=0
  i=1
1   y=y+x(i)*x(i)
    i=i+1
    if (i<n) goto 1
```

At first glance the loop had one fp add, one fp multiply, and one fp load. It also had one integer add, one integer comparison and one conditional branch. Unless the processor supports speculative loads, the loading of `x(i+1)` cannot start until the comparison completes.

Unrolling

```
y=0
do i=1,n-mod(n,2),2
  y=y+x(i)*x(i)+x(i+1)*x(i+1)
enddo
if (mod(n,2)==1) y=y+x(n)*x(n)
```

This now looks like

```
  y=0
  i=1
  n2=n-mod(n,2)
1   y=y+x(i)*x(i)+x(i+1)*x(i+1)
    i=i+2
    if (i<n2) goto 1
  if (mod(n,2)==1) y=y+x(n)*x(n)
```

The same 'loop overhead' of integer control instructions now deals with two iterations, and a small *coda* has been added to deal with odd loop counts. Rely on the compiler to do this.

The compiler will happily unroll to greater *depths* (2 here, often 4 or 8 in practice), and may be able to predict the optimum depth better than a human, because it is processor-specific.

Reduction

This dot-product loop has a nasty data dependency on `y`: no add may start until the preceeding add has completed. However, this can be improved:

```
t1=0 ; t2=0
do i=1,n-mod(n,2),2
  t1=t1+x(i)*x(i)
  t2=t2+x(i+1)*x(i+1)
enddo
y=t1+t2
if (mod(n,2)==1) y=y+x(n)*x(n)
```

There are no data dependencies between `t1` and `t2`. Again, rely on the compiler to do this.

This class of operations are called reduction operations for a 1-D object (a vector) is reduced to a scalar. The same sort of transform works for the sum or product of the elements, and finding the maximum or minimum element.

Reductions change the order of arithmetic operations and thus change the answer. Conservative compilers won't do this without encouragement.

Again one should rely on the compiler to do this transformation, because the number of partial sums needed on a modern processor for peak performance could be quite large, and you don't want your source code to become an unreadable lengthy mess which is optimised for one specific CPU.

Vectorisation

Assuming that the CPU has vectors with a length of four:

```
vtmp(1:4)=0
do i=1,n-mod(n,4),4
  vtmp(1:4)=vtmp(1:4)+x(i:i+3)*x(i:i+3)
enddo
y=sum(vtmp(1:4))
if (mod(n,4).ne.0) add on the odd iterations
```

Loading the four elements `x[i:i+3]` into a vector register is a single instruction, squaring all four elements is a single instruction, and adding them onto the vector `vtmp` (assumed to be held in a single vector register) is a single instruction.

This is how compilers generally vectorise: each element of the vector registers is processing a different iteration of the loop, and there must be no data dependencies between those iterations (save for reductions which can be eliminated).

In this case one might argue for additional registers to be used to accumulate the sum so that the vector instructions can be overlapped, e.g. a stride of 8 and two independent vector sums in each iteration.

More Vectorisation

```
complex (kind=kind(1d0)) :: z,z0

z=z0
do i=1,n
  z=z*z+z0
  if (abs(z).gt.4d0) exit
enddo
```

The core of a Mandelbrot Set generator. The loop has an unpredictable conditional exit and each iteration depends on the previous – not vectorisable in the manner described above.

But many compilers, if the target supports at least SSE3 (a minor, but important, addition to SSE2), will treat the complex datatype as a vector of two elements. So this may vectorise with a vector length of two.

The additions of SSE3 reduced the shuffling needed to perform complex-complex multiplication if both parts of a complex number were stored in the same vector register.

Prefetching

```
  y=0
  do i=1,n
    prefetch_to_cache x(i+8)
    y=y+x(i)*x(i)
  enddo
```

As neither C/C++ nor Fortran has a prefetch instruction in its standard, and not all CPUs support prefetching, one must rely on the compiler for this.

This works better after unrolling too, as only one prefetch per cache line is required. Determining how far ahead one should prefetch is awkward and processor-dependent.

It is possible to add directives to one's code to assist a particular compiler to get prefetching right: something for the desperate only.

Loop Elimination

```
do i=1,3
  a(i)=0
endo
```

will be transformed to

```
a(1)=0
a(2)=0
a(3)=0
```

Note this can only happen if the iteration count is small *and* known at compile time. Replacing '`3`' by '`n`' will cause the compiler to unroll the loop about 8 times, and will produce dire performance if `n` is always 3.

Loop Fusion

```
do i=1,n
  x(i)=i
enddo
do i=1,n
  y(i)=i
enddo
```

transforms trivially to

```
do i=1,n
  x(i)=i
  y(i)=i
enddo
```

eliminating loop overheads, and increasing scope for CSE. Good compilers can cope with this, a few cannot.

Assuming `x` and `y` are real, the implicit conversion of `i` from integer to real is a common operation which can be eliminated.

Fusion or Fission?

Ideally temporary values within the body of a loop, including pointers, values accumulating sums, etc., are stored in registers, and not read in and out on each iteration of the loop. A sane RISC CPU tends to have 32 general-purpose integer registers and 32 floating point registers.

Intel's 64 bit processors have just 16 integer registers, and 16 floating point vector registers storing two (or four in recent processors) values each. Code compiled for Intel's 32 bit processors uses just half this number of registers.

A 'register spill' occurs when a value which ideally would be kept in a register has to be written out to memory, and read in later, due to a shortage of registers. In rare cases, loop fission, splitting a loop into two, is preferable to avoid a spill.

Fission may also help hardware prefetchers spot memory access patterns.

Strength reduction

```
double a(2000,2000)

do j=1,n
  do i=1,n
    a(i,j)=x(i)*y(j)
  enddo
enddo
```

The problem here is finding where the element `a(i,j)` is in memory. The answer is $8(i-1)+16000(j-1)$ bytes beyond the first element of `a`: a hideously complicated expression.

Just adding eight to a pointer every time `i` increments in the inner loop is much faster, and called strength reduction. Rely on the compiler again.

Inlining

```
function norm(x)
double precision norm,x(3)

norm=x(1)**2+x(2)**2+x(3)**2
end function
...
a=norm(b)
```

transforms to

```
a=b(1)**2+b(2)**2+b(3)**2
```

eliminating the overhead of the function call.

If all function calls within a loop can be inlined, it may then mean that the loop can be vectorised, or otherwise more aggressively optimised.

Often only possible if the function and caller are compiled simultaneously.

Instruction scheduling and loop pipelining

A compiler ought to move instructions around, taking care not to change the resulting effect, in order to make best use of the CPU. It needs to ensure that latencies are 'hidden' by moving instructions with data dependencies on each other apart, and that as many instructions as possible can be done at once. This analysis is most simply applied to a single pass through a piece of code, and is called *code scheduling*.

With a loop, it is unnecessary to produce a set of instructions which do not do any processing of iteration `n+1` until all instructions relating to iteration `n` have finished. It may be better to start iteration `n+1` before iteration `n` has fully completed. Such an optimisation is called *loop pipelining* for obvious reasons..

Sun calls 'loop pipelining' 'modulo scheduling'.

Consider a piece of code containing three integer adds and three fp adds, all independent. Offered in that order to a CPU capable of one integer and one fp instruction per cycle, this would probably take five cycles to issue. If reordered as $3\times$(integer add, fp add), it would take just three cycles.

Debugging

The above optimisations should really never be done manually. A decade ago it might have been necessary. Now it has no beneficial effect, and makes code longer, less readable, and harder for the compiler to optimise!

However, one should be aware of the above optimisations, for they help to explain why line-numbers and variables reported by debuggers may not correspond closely to the original code. Compiling with all optimisation off is occassionally useful when debugging so that the above transformations do not occur.

Loop interchange

The conversion of

```
do i=1,n
  do j=1,n
    a(i,j)=0
  enddo
enddo
```

to

```
do j=1,n
  do i=1,n
    a(i,j)=0
  enddo
enddo
```

is one loop transformation most compilers do get right. There is still no excuse for writing the first version though.

The Compilers

```
f90 -fast -o myprog myprog.f90 func.o -lnag
```

That is options, source file for main program, other source files, other objects, libraries. Order does matter (to different extents with different compilers), and should not be done randomly.

Yet worse, random options whose function one cannot explain and which were dropped from the compiler's documentation two major releases ago should not occur at all!

The compile line is read from left to right. Trying

```
f90 -o myprog myprog.f90 func.o -lnag -fast
```

may well apply optimisation to nothing (i.e. to the source files following `-fast`). Similarly

```
f90 -o myprog myprog.f90 func.o -lnag -lcxml
```

will probably use routines from NAG rather than cxml if both contain the same routine. However,

```
f90 -o myprog -lcxml myprog.f90 func.o -lnag
```

may also favour NAG over cxml with some compilers.

Calling Compilers

Almost all UNIX commands never care about file names or extensions.

Compilers are very different. They do care greatly about file names, and they often use a strict left to right ordering of options.

Extension	File type
.a	static library
.c	C
.cc	C++
.cxx	C++
.C	C++
.f	Fixed format Fortran
.F	ditto, preprocess with cpp
.f90	Free format Fortran
.F90	ditto, preprocess with cpp
.i	C, do not preprocess
.o	object file
.s	assembler file

Consistency

It is usual to compile large programs by first compiling each separate source file to an object file, and then linking them together.

One must ensure that one's compilation options are consistent. In particular, one cannot compile some files in 32 bit mode, and others in 64 bit mode. It may not be possible to mix compilers either: certainly on our Linux machines one cannot link together things compiled with NAG's f95 compiler and Intel's ifort compiler.

Common compiler options

`-lfoo` and `-L`

`-lfoo` will look first for a shared library called `libfoo.so`, then a static library called `libfoo.a`, using a particular search path. One can add to the search path (`-L${HOME}/lib` or `-L.`) or specify a library explicitly like an object file, e.g. `/temp/libfoo.a`.

`-O`, `-O`**n** and `-fast`

Specify optimisation level, `-O0` being no optimisation. What happens at each level is compiler-dependent, and which level is achieved by not specifying `-O` at all, or just `-O` with no explicit level, is also compiler dependent. `-fast` requests fairly aggressive optimisation, including some unsafe but probably safe options, and probably tunes for specific processor used for the compile.

`-c` and `-S`

Compile to object file (`-c`) or assembler listing (`-S`): do not link.

`-g`

Include information about line numbers and variable names in `.o` file. Allows a debugger to be more friendly, and may turn off optimisation.

More compiler options

`-C`

Attempt to check array bounds on every array reference. Makes code much slower, but can catch some bugs. Fortran only.

`-r8`

The `-r8` option is entertaining: it promotes all single precision variables, constants and functions to double precision. Its use is unnecessary: code should not contain single precision arithmetic unless it was written for a certain Cray compiler which has been dead for years. So your code should give identical results whether compiled with this flag or not.

Does it? If not, you have a lurking reference to single precision arithmetic.

The rest

Options will exist for tuning for specific processors, warning about unused variables, reducing (slightly) the accuracy of maths to increase speed, aligning variables, etc. There is no standard for these.

IBM's equivalent of `-r8` is `-qautodbl=dbl4`.

A Compiler's view: Basic Blocks

A compiler will break source code into *basic blocks*. A basic block is a sequence of instructions with a single entry point and a single exit point. If any instruction in the sequence is executed, all must be executed precisely once.

Some statements result in multiple basic blocks. An if/then/else instruction will have (at least) three: the conditional expression, the then clause, and the else clause. The body of a simple loop may be a single basic block, provided that it contains no function calls or conditional statements.

Compilers can amuse themselves re-ordering instructions within a basic block (subject to a little care about dependencies). This may result in a slightly complicated correspondence between line numbers in the original source code and instructions in the compiled code. In turn, this makes debugging more exciting.

A Compiler's view: Sequence Points

A sequence point is a point in the source such that the consequences of everything before it point are completed before anything after it is executed. In any sane language the end of a statement is a sequence point, so

```
a=a+2
a=a*3
```

is unambiguous and equivalent to `a=(a+2)*3`.

Sequence points usually confuse C programmers, because the increment and decrement operators `++` and `--` do not introduce one, nor do the commas between function arguments.

```
j=(++i)*2+(++i);
printf("%d %d %d\n",++i,++i,++i);
```

could both do *anything*. With `i=3`, the first produces 13 with most compilers, but 15 with Open64 and PathScale. With `i=5`, the latter produces '`6 7 8`' with Intel's C compiler and '`8 8 8`' with Gnu's. Neither is wrong, for the subsequent behaviour of the code is completely undefined according to the C standard. No compiler tested produced a warning by default for this code.

And: there's more

```
if ((i>0)&&(1000/i)>1) ...

if ((i>0).and.(1000/i>1)) ...
```

The first line is valid, sane, C. In C `&&` is a sequence point, and logical operators guarantee to short-circuit. So in the expression

```
A&&B
```

A will be evaluated before B, and if A is false, B will not be evaluated at all.

In Fortran none of the above is true, and the code may fail with a division by zero error if `i=0`.

```
A.and.B
```

makes no guarantees about evaluation order, or in what circumstances both expressions will be evaluated.

What is true for `&&` in C is also true for `||` in C.

Fortran 90

Fortran 90 is *the* language for numerical computation. However, it is not perfect. In the next few slides are described some of its many imperfections.

Lest those using C, C++ and Mathematica feel they can laugh at this point, nearly everything that follows applies equally to C++ and Mathematica. The only (almost completely) safe language is F77, but that has other problems.

Most of F90's problems stem from its friendly high-level way of handling arrays and similar objects.

So that I am not accused of bias,

`http://www.tcm.phy.cam.ac.uk/~mjr/C/`

discusses why C is even worse…

Slow arrays

```
a=b+c
```

Humans do not give such a simple statement a second glace, quite forgetting that depending what those variables are, that could be an element-wise addition of arrays of several million elements. If so

```
do i=1,n
  a(i)=b(i)+c(i)
enddo
```

would confuse humans less, even though the first form is neater. Will both be treated equally by the compiler? They should be, but many early F90 compilers produce faster code for the second form.

Big surprises

```
a=b+c+d
```

really ought to be treated equivalently to

```
do i=1,n
  a(i)=b(i)+c(i)+d(i)
enddo
```

if all are vectors. Many early compilers would instead treat this as

```
temp_allocate(t(n))
do i=1,n
  t(i)=b(i)+c(i)
enddo
do i=1,n
  a(i)=t(i)+d(i)
enddo
```

This uses much more memory than the F77 form, and is much slower.

Sure surprises

```
a=matmul(b,matmul(c,d))
```

will be treated as

```
temp_allocate(t(n,n))
t=matmul(c,d)
a=matmul(b,t)
```

which uses more memory than one may first expect. And is the `matmul` the compiler uses as good as the `matmul` in the BLAS library? Not if it is Compaq's compiler.

I don't think Compaq is alone in being guilty of this stupidity. See IBM's `-qessl=yes` option...

Note that even `a=matmul(a,b)` needs a temporary array. The special case which does not is `a=matmul(b,c)`.

Slow Traces

```
integer, parameter :: nn=512

allocate (a(16384,16384))

call tr(a(1:nn,1:nn),nn,x)

subroutine tr(m,n,t)
double precision m(n,n),t
integer i,n

t=0
do i=1,n
  t=t+m(i,i)
enddo

end subroutine
```

As `nn` was increased by factors of two from 512 to 16384, the time in seconds to perform the trace was 3ms, 13ms, 50ms, 0.2s, 0.8s, 2ms.

Mixed Languages

The `tr` subroutine was written in perfectly reasonable Fortran 77. The call is perfectly reasonable Fortran 90. The mix is not reasonable.

The subroutine requires that the array it is passed is a contiguous 2D array. When `nn=1024` it requires `m(i,j)` to be stored at an offset of $8(i-1)+8192(j-1)$ from `m(1,1)`. The original layout of `a` in the calling routine of course has the offsets as $8(i-1)+131072(j-1)$.

The compiler must create a new, temporary array of the shape which `tr` expects, copy the relevant part of `a` into, and, after the call, copy it back, because in general a subroutine may alter any elements of any array it is passed.

Calculating a trace should be order n in time, and take no extra memory. This poor coding results in order n^2 in time, *and* n^2 in memory.

In the special case of `nn=16384` the compiler notices that the copy is unnecessary, as the original is the correct shape.

Bright people deliberate limit their stack sizes to a few MB (see the output of `ulimit -s`). Why? As soon as their compiler creates a large temporary array on the stack, their program will segfault, and they are thus warned that there is a performance issue which needs addressing.

Pure F90

```
use magic

call tr(a(1:nn,1:nn),nn,x)

module magic
contains
subroutine tr(m,n,t)
double precision m(:,:),t
integer i,n

t=0
do i=1,n
  t=t+m(i,i)
enddo

end subroutine
end module magic
```

This is decently fast, and does not make extra copies of the array.

Pure F77

```
allocate (a(16384,16384))

call tr(a,16384,nn,x)

subroutine tr(m,msize,n,t)
double precision m(msize,msize),t
integer i,n,msize

t=0
do i=1,n
  t=t+m(i,i)
enddo

end subroutine
```

That is how a pure F77 programmer would have written this. It is as fast as the pure F90 method (arguably marginally faster).

Type trouble

```
type electron
   integer :: spin
   real (kind(1d0)), dimension(3) :: x
end type electron

type(electron), allocatable :: e(:)
allocate (e(10000))
```

Good if one always wants the spin and position of the electron together. However, counting the net spin of this array

```
s=0
do i=1,n
  s=s+e(i)%spin
enddo
```

is now slow, as an electron will contain 4 bytes of spin, 4 bytes of padding, and three 8 byte doubles, so using a separate spin array so that memory access was unit stride again could be eight times faster. Vectorisation will also be poorer, as a vector of non-consecutive spins cannot be loaded in a single operation.

What is `temp_allocate`?

Ideally, an `allocate` and `deallocate` if the object is 'large', and placed on the stack otherwise, as stack allocation is faster, but stacks are small and never shrink. Ideally reused as well.

```
a=matmul(a,b)
c=matmul(c,d)
```

should look like

```
temp_allocate(t(n,n))
t=matmul(a,b)
a=t
temp_deallocate(t)
temp_allocate(t(m,m))
t=matmul(c,d)
c=t
temp_deallocate(t)
```

with further optimisation if `m=n`. Some early F90 compilers would allocate all temporaries at the beginning of a subroutine, use each once only, and deallocate them at the end.

`a=sum(x*x)`

```
temp_allocate(t(n))
do i=1,n
  t(i)=x(i)*x(i)
enddo
a=0
do i=1,n
  a=a+t(i)
enddo
```

or

```
a=0
do i=1,n
  a=a+x(i)*x(i)
enddo
```

Same number of arithmetic operations, but the first has $2n$ reads and n writes to memory, the second n reads and no writes (assuming `a` is held in a register in both cases). Use `a=dot_product(x,x)` not `a=sum(x*x)`! Note that a compiler good at loop fusion may rescue this code.

Universality

The above examples pick holes in Fortran 90's array operations. This is not an attack on F90 – its array syntax is very convenient for scientific programming. It is a warning that applies to all languages which support this type of syntax, including Matlab, Python, C++ with suitable overloading, etc.

It is not to say that all languages get all examples wrong. It is to say that most languages get some examples wrong, and, in terms of efficiency, wrong can easily cost a factor of two in time, and a large block of memory too. Whether something is correctly optimised may well depend on the precise version of the compiler / interpreter used.

Fortran being right

```
do concurrent (i=1:n)
  a(i)=b(i)+c(i)
enddo
```

'`do concurrent`' is a guarantee from the programmer that the loop iterations are independent, and can be executed in any order. Furthermore, the value of the loop counter after the loop exits is undefined.

This was introduced in the 2008 Fortran standard, and aids vectorisation and auto-parallelisation.

Of course this trivial example ought to be equally efficient written with Fortran's standard array syntax

```
a=b+c
```

or, if only part of the arrays are being updated

```
a(1:n)=b(1:n)+c(1:n)
```

Ought.

Intel’s Evolution

CPU families

This discourse on the history of Intel CPUs serves two puposes. One is to be somewhat relevant – one cannot deny that you are almost certain to end up running programs on computers with Intel processors. The other is to demonstrate that not all CPUs are equal, and design choices need to be made.

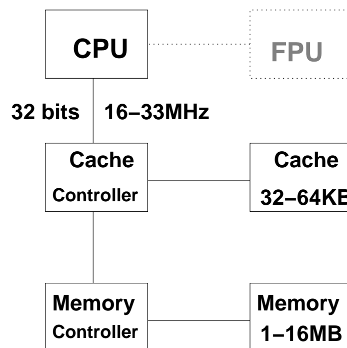
The 80386 (i386)

This CPU was launched in 1985, and first appeared in a PC in 1987. It is the oldest we will consider in detail.

It was Intel's first 32 bit processor, capable of 32 bit addressing and arithmetic operations on 32 bit integers.

It supported virtual memory, paging, and the concept of a privileged mode of operation. It was the first Intel processor capable of running UNIX (or any other modern operating system) in a sensible manner.

It had eight not quite general purpose integer registers, and no floating point support at all. For that one needed to procure a separate floating point unit.



The i386: what was missing?

No cache.

No instruction re-ordering (OOO).

Not superscalar (but mostly pipelined, theoretically issuing one instruction every other clock cycle).

No register renaming.

Clock speed the same as its external databus (16MHz to 33MHz).

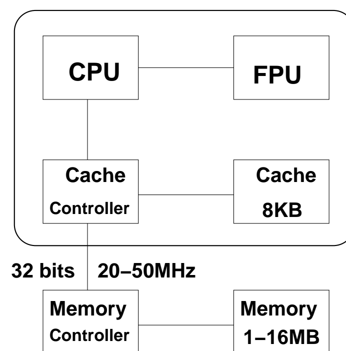
Optional floating point unit is not pipelined at all, and very slow, with an addition taking about 20 clock cycles. It was incapable of reaching 1MFLOPS on Linpack.

The 80486 (i486)

Released in 1989, depending on one's point of view, this was either a non-event, or a huge leap forwards. It offered no new capabilities, save that it combined a 80386, its optional floating point unit, its optional external write-through cache controller, and 8KB of cache, all on to a single die, which contained 1.2 million transistors, over four times as many as the i386.

The integer unit was now fully pipelined, capable of issuing one instruction every clock cycle.

The floating point unit was still not pipelined, but simple operations now executed about twice as fast. With clock speeds of 20MHz to 50MHz, it could just about achieve 1MFLOPS on Linpack.



i486DX2 and DX4

Because the i486 had on-chip cache, it was sensible to increase its clock speed beyond that of the external bus. As long as the cache hit rate was good, then the slow speed of the external bus would not hold back the CPU.

So the i486DX2 ran at twice the speed of the external bus (25/50MHz or 33/66MHz), and the later i486DX4 at three times the speed of the external bus (25/75MHz or 33/100MHz). In order to improve its cache hit rate, the DX4 had a 16KB cache.

For the first time heat dissipation became an issue. Even the DX2 at 5V (the traditional voltage for logic circuits) and 66MHz produced only about 6W, and barely needed a fan, but certainly needed a heatsink. The DX4 required a 3.3V supply, and, in many designs, a fan too.

The Pentium

The Pentium was released in 1993. It was a complete redesign of the i486, although still used the same instruction set, so codes did not need to be recompiled.

The floating point unit was now fully pipelined, and the cache was now split into separate instruction and data caches.

It was the last Intel CPU to be introduced running at the same speed as its external bus (60 and 66MHz), and running at 5V. Its power dissipation was about 15W – it certainly needed a fan.

It was the first Intel CPU to support half integer clock multipliers, with varients running at up to 200MHz. All those running at more than 66MHz ran at 3.3V.

Not only was the bus speed doubled from the i486DX4 from 33MHz to 66MHz, but the bus width was doubled from 32 bits to 64 bits, where it would remain for a very long time.

Superscalar

The Pentium had two separate instruction pipelines, called U and V. One could accept any instruction, the other a small subset of possible instructions. But, under the right conditions, instructions could pair and be issued to both pipes simultaneously.

So the Pentium was the first superscalar CPU in Intel's x86 range. To help maintain its pipeline, it had a branch predictor, which the i486 lacked.

The Pentium had 3.1 million transistors, over ten times as many as the i386. But even if run at the same clock speed it would offer a large performance increase over the i386, most significantly in floating point throughput, which might be up to twenty times higher, but even in integer instructions where the i386 could manage a theoretical peak of one instruction every other clock cycle, and the Pentium two instructions per clock cycle.

Not only that, but the first revision of the Pentium supported two processors in a single computer: SMP.

A Name, not a Number

Intel discovered one could not trademark numbers, but could trademark names. So it is the Pentium™.

Also of political interest, both IBM and AMD had a licence which enabled them to fabricate and sell Intel processors from the 8086 to the i486 fairly freely. This licence did not extend to the Pentium.

Both AMD and IBM were quite good at taking Intel's designs, tweaking them slightly to improve them, and then making and selling their own versions.

MMX

The Pentium MMX was a minor revision to the Pentium which included Multi Media eXtensions. These reused the large floating-point registers to store vectors of integer data, and introduced operations to act on all elements of those vectors simultaneously.

The target applications were photographic filters (e.g. Photoshop), and graphics effects in games. The idea of add-with-saturate as a single instruction was introduced, which is very useful in graphics calculations. It simply says that if trying to combine two sources into a single 8 bit pixel, one of a brightness of 200, one of 120, then the answer is 255 (maximum possible 8-bit value), and not 64 (addition with conventional wrap-around).

A total of 57 new instructions were added, packing the floating point registers with eight 8 bit integers, four 16 bit integers, two 32 bit integers, or one 64 bit integer.

MMX trouble

MMX presented Intel with two problems. Firstly, it introduced new instructions. Old code compiled with old compilers would make no reference to those instructions, not use them and go no faster.

Secondly, the new instructions were not ones which it was easy for a compiler to spot and issue.

```
unsigned char *a,*b,*c;

for (i=0;i<8;i++){
  c[i]=a[i]+b[i];
  if (c[i]<a[i]) c[i]=255;
}
```

That loop is something like a single MMX instruction, provided that the compiler spots it. Arguably

```
for (i=0;i<8;i++)
  a[i]=((a[i]+b[i])>a[i])?(a[i]+b[i]):255;
```

would be an even closer match.

The Answer

'Buy our new processor: your existing code will run no faster' is a hard problem, even without the 'P.S. Would you like to code in assembler, because life is really difficult for our compiler team?'

Intel's solution was two-fold. Firstly, it encouraged some key applications to adopt MMX rapidly, and to publish wonderful benchmarks on the speedups of factors of three and more.

Secondly, it significantly improved the branch predictor and the L1 caches on the Pentium MMX. By moving from 8KB 2-way associative to 16KB 4-way associative, hit rates improved, and most code saw a speed improvement of 5-10% without recompilation, even if this improvement was nothing to do with the presence of the MMX instructions.

The 8KB cache on the old i486DX2 had been 4-way associative too. With the plain Pentium's cache just 2-way associative, it was easy to write code which would run faster on the DX2 by deliberately using an access pattern cacheable with 4-way associativity, but not with two-way. The Pentium MMX stopped this trick.

The Pentium Pro

The Pentium Pro, introduced in 1995, is an often forgotten member of Intel's family. Its successor arrived only 18 months later, and in many ways the Pentium Pro was a step backwards from the Pentium MMX: it lacked the MMX instructions, and its L1 cache size went back to 8KB. Clock speeds varied from 150MHz to 200MHz.

However, it did make two significant advances which all future Intel processors would share.

The L2 cache was now integrated onto the chip (but not yet the die). This meant it could be much faster, as signals did not need to take a long, complicated path from chip through socket onto motherboard with all the potential for electrical interference that implies. It also kept L2 cache traffic off the bus from the processor to the motherboard. The L2 cache size was usually 256KB or 512KB.

RISC translation

The second major advance of the Pentium Pro was that the core of the CPU was now a RISC core. The x86 instructions were translated into RISC instructions, called micro ops, and then these fed into a RISC core with its instruction issuing logic. The RISC core was a superscalar out-of-order core.

This design leads to a longer instruction pipeline than would be the case for a pure RISC design. The x86 instructions are first decoded into microcode instructions, and then these pass through a second decoding stage, followed by reordering, as expected for a standard RISC core. The Pentium Pro design can decode three x86 instructions per clock cycle, and issue five μ-ops per clock cycle. A single x86 instruction typically produces between one and three μ-ops, but can produce considerably more.

Pure RISC: Fetch, decode, queue/reorder, execute

PPro: Fetch, translate to μ-ops, fetch, decode, queue/reorder, execute

Pentium II

The Pentium II, introduced in 1997, was a relatively small change from the Pentium Pro. It added the MMX instructions and returned the L1 cache sizes to 16KB each, which removed any advantage that the Pentium MMX had.

It was packaged in a form which fitted into a slot, rather than a socket, which made it easier for Intel to make as it consisted of a processor die (of about 7.5m transistors) and again off-die L2 cache. This sat on what was effectively a tiny circuit-board. The Pentium II ran its L2 cache at just half the CPU speed, rather than full speed as the Pentium Pro had done. On the other hand, it typically had twice as much.

Variants ran at from 233MHz to 333MHz with a 66MHz bus, and later from 300MHz to 450MHz with a 100MHz bus.

Pentium III

The Pentium III was another minor update to the Pentium Pro design. Initial versions continued with a 100MHz bus, and then later versions moved to 133MHz. Clock speeds ranged from 450MHz to 1.4GHz.

Improvements in process technology during the Pentium III's life enabled Intel to move the L2 cache on die. Most Pentium IIIs with on-die cache had just 256KB, which needed more transistors than the rest of the CPU (including its L1 caches).

Pentium III: SSE

One important change with the Pentium III was the introduction of the SSE instructions. These added eight new 128-bit registers, each containing four single-precision floating point numbers. These new registers required OS support, as they needed to be saved when task switches occured, and the amount of saved state for a process is now 128 bytes bigger.

The instructions were designed for 3D games: they require floating point calculations to perform projections, but do not require the precision that most scientific calculations need. For code which could make use of them, they certainly increased performance, as a full vector of four adds could be started every other clock cycle, and independently four multiplies every other cycle.

Pentium 4

The Pentium 4 architecture, introduced in late 2000, was in many ways revolutionary.

Of most immediate interest to scientists, it introduced SSE2. This extended SSE to operate on vectors of two double precision numbers. At last useful for general purpose scientific computing! In a similar fashion to the Pentium III, its execution unit took two clock cycles to absorb a two element double precision add or multiply.

The changes in design were many. In no way was the Pentium 4 part of a continuous development from the Pentium Pro. The idea now was to move to high clock speeds partly by reducing the amount done per clock cycle.

As a minor change, the Pentium 4 returned to a traditional socket shape, rather than the slot of the Pentium II and III. Its L2 cache was on die, so the idea of a processor being a mini motherboard was no longer needed.

Caching

The Pentium 4 cached not x86 instructions, but rather the μ-ops generated from them. It could cache about 12,000 μ-ops, so, in most loops the overhead of continuously decoding x86 instructions to μ-ops was eliminated on all but the first iteration.

The length of the instruction pipeline, about 10 stages in the Pentium III, became 20 stages. Fortunately the branch predictor was improved, for a mispredicted branch was rather expensive. Later Pentium 4's, the so-called 'Prescott' revision, increased the pipeline length to 30 stages.

Clock Speeds

The Pentium 4 was introduced at 1.4GHz, at which speed it was not very competitive with the faster Pentium IIIs. By the end of 2001 it reached 2GHz, then in 2002 3GHz versions were released. The final speed reached in 2004 was 3.8GHz, though at the cost of needing to dissipate 115W.

Given such high clock speeds, it is extraordinary that part of the integer unit ran at twice the main clock speed. This enabled two data-dependent integer operations to execute in a single clock cycle (for certain combinations of operation).

Writing a (pointless) benchmark which will execute faster on a Pentium 4 from 2004 than any current Intel processor is not hard.

AMD

Whilst the Pentium 4 was Intel's main processor, Intel's rival, AMD, produced the Opteron (and Athlon64) in 2003.

These produced two significant advantages over Intel's Pentium 4, and earlier AMD processors. Firstly the memory controller was integrated onto the CPU die, with the CPU connected directly to the memory DIMMs, and memory traffic no longer on the same bus as I/O (PCI devices). This produced a significant performance increase. The original memory interface on the Opteron was 6.4GB/s (128 bit, 400MT/s), similar to the shared I/O and memory interface on the Pentium 4 (64 bit, 400 to 1066MT/s), but with lower latency.

AMD64

AMD's big change with the Opteron was to extend Intel's 32 bit architecture to 64 bits. This was done in a particularly wholesale manner, with many significant changes made which had nothing directly to do with the move from 32 bits to 64.

The eight 32-bit integer registers were extended to sixteen 64-bit ones.

The eight SSE registers were also doubled in number to sixteen.

The name was a problem. Although Intel adopted AMD's instruction set, it was never going to call it AMD64, preferring EM64T and then Intel 64. More neutral observers settled on x86-64 (or x86_64).

Intel's first response to AMD's 64 bit coup was to develop the Pentium4 EM64T, which supported the same 64 bit instructions. Intel also slightly extended SSE2 with SSE3, adding a few new instructions, notably "horizontal" operations, so rather than adding the vectors `v` and `u` returning a vector of `v[0]+u[0]` and `v[1]+u[1]` it would return `v[0]+v[1]` and `u[0]+u[1]`, and also a mixed addition and subtraction useful in complex multiplication. It returns `v[0]+u[0]` and `v[1]-u[1]`.

Multiple Cores

Intel and AMD were both having difficulty increasing clock speeds, and yet no difficulty fitting more transistors on wafers. So the obvious answer was to move to dual core CPUs. Intel introduced the dual core Pentium D (based on the Pentium 4) in 2005, beating AMD Athlon64 X2 by a few weeks, but a few weeks behind AMD's dual core Opterons intended for the server market.

Intel's solution was less elegant – two dies in the same chip package, whereas AMD used a single die.

Goodbye, Pentium 4

Intel's next processor, the Core, introduced in 2006, abandoned the architecture of the Pentium 4, and looked much more like a highly modified Pentium III. It did keep something very similar to the Pentium 4's bus.

The Pentium 4 had proven to be very power hungry, and Intel had not been able to get it up to the clock speeds it had hoped. The Pentium M, another development of the Pentium III, had been produced in 2004 for notebooks, offering better performance per watt than the Pentium 4, and the Core was an extension of this.

Surprisingly the Core remained 32 bit only – it did not support the EM64T extensions.

Core 2

The Core 2 was released only about seven months after the Core, and it regained the 64 bit extensions. It was the first Intel processor not to have a single core varient for desktops – it was introduced as a dual core processor, and soon moved to quad core.

Its clock speeds were more modest than the Pentium 4, ranging from about 1.8GHz to 3.3GHz. It was a substantial evolution of the old Pentium Pro architecture, and it retained the bus of the Pentium 4.

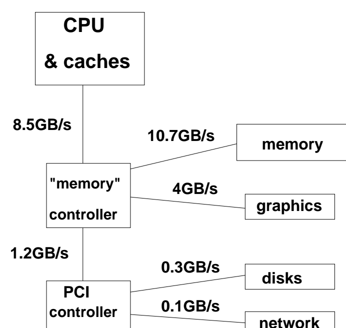
In terms of theoretical peak performance, each core could now execute two element add and multiply instructions independently on every clock cycle, so four FLOPS per Hz – twice performance of the Pentium 4. It also increased the size of the L1 caches to 32KB each.

Slow buses

The single external bus on the Pentium 4 and Core / Core2, carried all memory and I/O traffic (disk, network, graphics card), and was 64 bits wide running at a range of disappointing speeds.

Disappointing? The dual core 2.13GHz Core 2 E6420 I have in front of me ran it at 1066MT/s. It took dual channel DDR2/667 memory, so the memory was capable of providing 10.67GB/s, but the bus carrying all memory and I/O traffic to the CPU was capable of just 8.5GB/s. Later varients increased the bus speed to 1333MT/s, but also increased the memory speed to 800MT/s.

The last Core 2s moved to using DDR3/1066 memory, but with the CPU bus still running at just 1333MT/s. So 17GB/s of potential from the memory being throttled to 10.67GB/s by the bus to the CPU, and this one bus was shared by all CPU cores. (E.g. 2.83GHz quad core Q9550 Core 2, launched in 2008.)



And Slower

Dual socket Core 2 designs were yet worse. Because both CPUs shared the same bus connecting them to each other and the combined memory and I/O controller, the bus was physically longer with more connections than on a single socket machine. So it tended to run slightly slower.

So a dual socket Core 2 based machine often had less total memory bandwidth than a single socket Core 2 based machine, which was in turn in general beaten by AMD's Athlon64 which had placed the memory controller on the processor die with the memory connecting directly to it, and a separate I/O bus.

A dual socket Core 2 machine I had access to at the time had four channels of DDR2/667 memory – 21.3GB/s – but feeding into a 1333MT/s bus capable of just 10.67GB/s. It is little wonder that a standard memory benchmark achieved just 6.4GB/s.

The History of Buses

Intel had long used a shared bus for memory and I/O. On the first PC the bus which was exposed for plugging expansion cards into (graphics cards, disk controllers, etc) was very similar to the CPU's bus: 8 bits wide, 4.77MHz, and one data transfer every four clock cycles.

The 80286-based PCs improved this to 16 bits wide, 8MHz and one data transfer every three clock cycles. This new bus could still accept old cards, provided they were happy to run at 8MHz, which most were. They would still suffer 8 bit four cycle transfers.

The the i386 arrived, and there was chaos. IBM produced a very nice, very expensive, bus completely incompatible with the previous 'ISA' bus. So other vendors extended the ISA bus again, to produce something rather better, and still compatible with old cards, called EISA.

PCI

Eventually, in 1992, in time for the Pentium, the PCI bus was created. This was an enormous leap from the 16 bit ISA bus which was still widely used: PCI was 32 bit, 33MHz, and four data transfers in five clock cycles. A change from 5.3MB/s to 105MB/s. For the first couple of years little could take full advantage of that potential.

PCI was revolutionary in other ways too. Before PCI, buses tended to be closely tied to one's processor. Intel CPUs used ISA, Motorola (in Macs) NuBus, SPARC in Sun SBUS, Alpha (DEC) TurboChannel. PCI was much more processor-neutral, and soon everyone was using it.

But PCI was still a traditional bus in the sense of being electrically a single bus to which multiple devices could be added. This causes electrical difficulties at high speeds as impedence matching becomes important and impossible.

AGP and PCIe

Graphics cards provided the greatest momentum for faster buses, and AGP was the initial answer. It was quite similar to PCI, but, by being a point-to-point bus, it was able to use much higher clock speeds. PCI did ultimately experiment with 66MHz operation, but AGP used data transfer rates of 133MT/s to 533MT/s.

Then, in about 2004, the PCIe bus emerged as the successor to PCI. By being point-to-point it was able to use high signalling rates: 2.5GT/s when introduced, doubled to 5GT/s in 2007. It is very narrow – interfaces a single bit wide are used and useful (500MB/s for PCIe v2, so fine for USB, 1GBit/s ethernet, and much else besides), but $\times 4$ (4 bits wide) are common (2GB/s, quite fast enough for disk controllers and 10GBit/s network interfaces), as are $\times 8$ and $\times 16$. The latter are used for graphics cards, co-procesors cards, and very high speed interconnects.

PCIe v3 (late 2011) is nearly twice as fast again.

PCIe Fun

A typical computer may have a mixture of widths of PCIe slots: $\times1$, $\times4$, $\times8$, $\times16$.

Narrow cards should fit into wider slots and work perfectly.

Wide cards might not mechanically fit into narrower slots, although sometimes the slots are made so that this is possible.

More confusingly, a slot which is physically the expected width for, say, a $\times8$ slot may be wired electrically as something narrower. Given that a $\times8$ card should work perfectly, even if the slot is wired $\times4$, one might not notice.

```
# lspci -vv
01:00.0 VGA compatible controller: [...]
          LnkCap: Port #0, Speed 5GT/s, Width x16 [...]
          LnkSta: Speed 5GT/s, Width x8 [...]
```

A $\times16$ PCIe v2 video card in a slot which is physically $\times16$ and electrically $\times8$.

LnkCap: Link capability
LnkSta: Link status

Nehalem

Intel's next CPU, codenamed Nehalem, was introduced in 2008. Its big advance was to move the memory controller onto the CPU die, as AMD had done three years earlier. Many varients had a three channel memory controller, rather than the usual two channel.

The other feature of the separate 'memory' controller (also known as the north bridge) of the Core2 was to provide a fast PCIe bus to the graphics card. This feature was also incorporated directly into the Nehalem CPU. However, a PCIe controller with just 16 PCIe lanes is built into the CPU. There is still an external PCIe controller, connected via a different interface, for other PCIe things (disks controllers, network controllers, etc.)

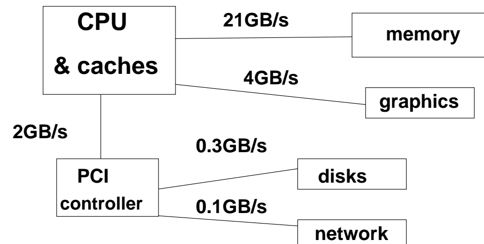
The biggest jump was for dual socket machines, which now had two independent memory buses. The Nehalem equivalent of the dual socket Core 2 machine on the previous slide had six channels of DDR3/800 memory. Theoretically 38.4GB/s, with a measured performance of 26GB/s, over four times what the Core 2 based machine achieved.

The peak floating point performance remained four FLOPS per Hz, but in general the Nehalem performed much better, mostly due to its increased memory bandwidth.

Other Nehalem changes

The Nehalem introduced the idea of an smallish (256KB) L2 cache for each core, unified between instruction and data, with the L3 cache being shared by all cores.

It also started to lengthen the pipeline again. This had reached 30 stages with the Pentium 4, dropped to about a dozen with the Core 2, and now rose to around twenty.



AVX and AVX2

The next major change after Nehalem was the introduction of AVX. This doubled the size of the vector registers, and the associated functional units, so that they could operate on 256-bit vectors, rather than 128-bit ones. Floating point vector instructions could now operate on 1, 2 or 4 doubles, or 1, 2, 4 or 8 singles.

AVX was introduced with the Sandy Bridge Core, and Ivybridge was a fairly minor redesign. Assuming one used the new instructions, and could use a vector length of four, the peak performance was now eight FLOPS per Hz per core.

The next redesign, Haswell and Broadwell, introduced a fused multiply-add (FMA) instruction (`a*b+c` as a single instruction), and could issue two of these every clock cycle, so a peak of sixteen FLOPS per Hz per core.

Lakes

Intel's current (2018) CPUs, Skylake and Kaby Lake, are very confusing.

Intel has again doubled the vector length to introduce AVX-512. But it has done so in a very fragmented fashion.

Laptop and most desktop CPUs simply do not support AVX-512 at all.

Some low-end server CPUs support the instructions, but have no dedicated AVX-512 units. Their peak FLOPS per Hz per core remains sixteen, and any AVX-512 instructions must pass through both AVX-256 units.

The high-end CPUs do have a pair of genuine AVX-512 execution units, and thus can achieve 32 FLOPS per Hz per core.

Some, but not all, of the CPUs with two AVX-512 units need to reduce their clock-speed to below their 'headline' speed when these are active.

Compatibility

If one runs code compatible for the Nehalem (or earlier) on a Haswell, it will not achieve more than four FLOPS per Hz. The only way of achieving more is to use the extra length in the vector registers, and to use the new FMA instruction.

There are three choices: abandon performance, abandon compatibility with older processors, or embrace code which executes different instructions depending on which processor it is running on. The latter route is sanest, but can lead to different bugs appearing depending on the processor one executes on.

Shared Libraries

Ideally everyone using Linux would agree on one set of libraries to use to cover the common, speed-critical, operations such as FFTs and Lapack. By linking these as shared libraries, one can then rely on the machine on which the code is executed having a version optimised for it which will be called automatically.

Unfortunately things don't quite work like this, especially as the maths libraries produced by Intel, which are reasonably fast, are not freely distributed, and have themselves had bugs in the past.

It is therefore quite likely that software purchased/installed a couple of years ago does not make full use of the latest processors. This is particularly true for AMD-based computers.

Sockets

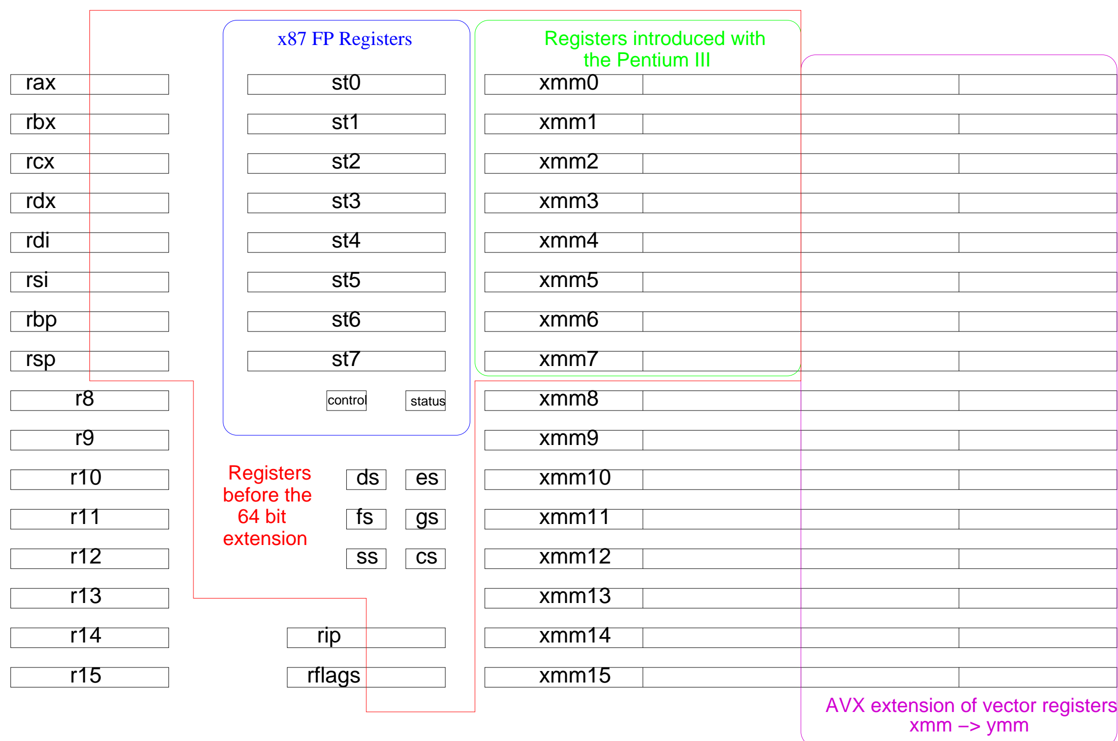
Intel's 'standard' uniprocessor CPUs currently use sockets of around 1150 pins. These have a 128 bit memory bus, and around 16 PCIe lanes into the lower-latency PCIe controller integrated onto the CPU.

Thus the Xeon E3-12xx and most Core-i7 processors.

There are rarer, high-end varients using sockets of around 2,000 pins. These have 256 bit memory buses, and 40 or more PCIe lanes. Thus the Xeon E5-16xx, Core i7 Extremes, and a few others. This configuration is much more common with dual socket machines (Xeon E5 and E7).

The high number of PCIe lanes going to the low-latency on-CPU PCIe controller is wanted by gamers using two video cards, and HPC people wanting to use two GPU cards, or a GPU card and an Infiniband card. The doubled memory bandwidth is wanted by almost anyone once one has more than four cores per CPU, as most of these CPUs do.

Intel Registers



Early Intel 32 bit processors

Feature	i386	i486	Pentium	P MMX
32 bit	Y	Y	Y	Y
FPU	option	Y	Y	Y
L1 cache	no	8KB	8KB+8KB	16KB+16KB
L2 cache	N	N	N	N
Instr per clock	0.5	1	2	2
FLOPS per Hz	c.0.04	c.0.08	1	1
branch predict	N	N	Y	Y
MMX	N	N	N	Y
Clock, MHz	16–33	25–100	60–200	150–233

Late C20th Intel 32 bit processors

Feature	P Pro	P II / P III	P4 / P4 EMT64
64 bit	N	N	N / Y
L1 cache	8KB+8KB	16KB+16KB	12Kμ-op+16KB
L2 cache	off die	on die	on die
MMX	N	Y	Y
SSE	N	N / Y	Y
SSE2	N	N	Y
FLOPS per Hz	1	1	2
Cores	1	1	1 or 2
Clock, MHz	150–200	233–450 / 450–1,400	1,500–3,600

All with RISC core fed by microcode decoder.

21st Century Intel Processors

Feature	Core 2	Nehalem	Sandy Bridge	Haswell	Skylake
Introduced	2006	2008	2011	2013	2015
Mem ctrl	N	Y	Y	Y	Y
Cache levels	2	3	3	3	3
Vector length	2	2	4	4	4
FMA	N	N	N	Y	Y
FLOPS per Hz	4	4	8	16	16
Memory	–	DDR3/1333	DDR3/1600	DDR3/1600	DDR4/2133
Clock, GHz	1.6–3.3	1.8–3.3	1.8–3.6	2.0–4.0	1.8–4.0

All 64 bit. All multi-core. All 32KB+32KB L1 cache. All save Core 2 256KB of L2 cache per core. All last level caches shared by all cores.

Some server variants of Skylake have a vector length of eight and support 32 FLOPS per Hz.

Some variants of Haswell support DDR4/2133.

-r8, 215
/proc, 185
0x, 60

address lines, 46, 47
AGP, 261
alignment, 128, 129
allocate, 174
AMD, 31, 132, 137
ARM, 31
ATE, 70
AVX, 130, 265

basic block, 216
BLAS, 84
bss, 172

C, 219
cache
- anti-thrashing entry, 70
- associative, 69, 73
- direct mapped, 66
- Harvard architecture, 74
- hierarchy, 71
- line, 63
- LRU, 73
- memory, 56, 58, 61
- write back, 72–74
- write through, 72

cache coherency
- snoopy, 72

cache controller, 59
cache thrashing, 68
CISC, 29, 33
clock, 17, 77, 78, 133, 134
compiler, 211–215
compilers, 32
cooling, 78
CPU family, 31
CSE, 195
CUDA, 149

data dependency, 22, 23
data segment, 172
debugging, 209, 214, 215
dirty bit, 72
disk thrashing, 162
division
- floating point, 37

DRAM, 45–49, 52, 56, 57
DTLB, 160

ECC, 53–55, 74

F90, 219
F90 mixed with F77, 224
fetch, decode, execute, 17
flash RAM, 45
FMA, 265
foolishness, 150
FPU, 16
free, 166
functional unit, 16, 21

GPUs, 140–150

heap, 172–174, 177–179
hex, 60
hit rate, 58, 70
hyperthreading, 135–139

ILP, 30
inlining, 207
instruction, 18, 19
instruction decoder, 16
instruction fetcher, 16
Intel, 31, 130, 132, 137
issue rate, 21
ITLB, 160

latency, functional unit, 21
libraries, shared, 175

linking, 214
Linpack, 40
loop
- blocking, 96, 97, 112–115
- coda, 198
- elimination, 109, 203
- fission, 205
- fusion, 204, 229
- interchange, 210
- invariant removal, 195
- pipelining, 208
- reduction, 199
- strength reduction, 206
- unrolling, 94, 198

malloc, 174
matrix multiplication, 83, 84
MFLOPS, 39
micro-op, 33
microcode, 36
MIPS, 31, 39
mmap, 173, 176–179

null pointer dereferencing, 173
Nvidia, 141–149
Nvidia Pascal, 142, 143

OpenMP, 149, 150
operating system, 162
optimisation, 193

page, 167–169
page colouring, 170
page fault, 162
page table, 158, 159, 161
pages, 154–156
- locked, 163

paging, 162
parity, 53, 74
PCI, 259, 260
PCIe, 148, 261, 262
physical address, 155
physical address, 154, 156
physical memory, 164
pipeline, 19–21
pipeline depth, 19
power, 78, 79
prefetching, 75, 76, 202
ps, 164, 165

register, 16
- architectural, 24
- physical, 24

register spill, 102, 205
registers, 205
RISC, 29, 33, 36

SDRAM, 48, 52
- timings, 49

segment, 172
segmentation fault, 156, 177, 180
sequence point, 217, 218
SMP, 239
SPEC, 41
speculative execution, 27
SRAM, 45, 56, 57
SSE, 130
stack, 172–174, 184
streaming, 76
streaming multiprocessor, 141, 142
superscalar, 29

T_{CAS}, 49
T_{RCD}, 49
T_{RP}, 49
tag, 61–67
text segment, 172
TLB, 157, 160, 161, 167
trace (of matrix), 223–226

ulimit, 165, 174

vector computer, 38
vectorisation, 122–134, 200, 201, 207, 227
virtual address, 154–156
virtual memory, 162, 164
voltage, 78

warp, 141–144

x87, 36, 127

Bibliography

Computer Architecture, A Qualitative Approach, 5th Ed., Hennessy, JL and Patterson, DA, pub. Morgan Kaufmann, c.£40.
Usually considered the standard textbook on computer architecture, and kept reasonably up-to-date. The fifth edition was published in 2011, although much material in earlier editions is still relevant, and early editions have more on paper, and less on CD / online, though with 850 pages, there is quite a lot on paper…