4. Code optimisation

> *– "Tell me Johnny, how come you always beat that rabbit?"*
> *– "Well, I'll tell you. The secret is modern design... streamlining! Yup, we turtles are built for racing. See for yourself - we got an airflow chassis. Now take rabbits. They're built all wrong for racing - those ridiculous ears! Wind resistance, son, just wind resistance." – Tortoise Wins by a Hare (1943)*

If a scientific algorithm (e.g. tomography reconstruction or image analysis) can be made to run in less time, then it can be run more frequently, giving better temporal resolution for real-time analysis or a higher throughput for processing offline data. Alternatively, the faster, "optimised" version can process larger datasets, in the same amount of time that the original algorithm takes to process a smaller dataset – allowing more precise results or more advanced modelling.

There are many ways to encode even the simplest arithmetic operations into a form that a computer can execute; some will execute faster than others and those faster-running implementations will often take longer to code. For example, take the following Javascript snippet:

```
var a = 2;          //Assign value of 2 to variable a
var b = 2;          //Assign value of 2 to variable b
var c = a + b;      //Add values of variables a and b, store to c
```

While the purpose of this script is immediately apparent, the means to executing it are not. Firstly, Javascript is an *interpreted* language; i.e. when run, the above code is translated on the fly by some program (the "interpreter") to instructions that the processor's electrical logic can understand. The addition operation will typically be executed on an *arithmetic logic unit* (ALU) within the processor. This need for the code to be *interpreted* and translated to "machine code" every time it is to be run delays the actual execution of the code somewhat each time. Scientific code is commonly written in *native* languages, which are translated to machine code (compiled) once, this machine code can then subsequently be run directly on the processor many times without the need for any intermediate interpreter stage. Optimising analysis-performing routines in a control system can decrease control lag, reducing the chance of unwanted oscillations within the control system.

4.1. Life of an instruction

In order to understand how common optimisations work, a simplified explanation of how a typical computer works is provided. The key parts of this explanation

(regarding optimisation) are execution units, the cache and the pipeline. Some aspects of processor architecture will not be covered, including RISC/CISC, out-of-order execution, speculative execution, branch prediction and cache hierarchy. The pipeline has also been greatly simplified; the instruction decode stage has been completely separated from the pipeline in the examples given in this chapter.

4.1.1. Storage, fetch and μops

When a program is run on a typical PC, the machine-code is initially loaded into system memory (RAM). The processor then *fetches* code and data from RAM as it executes. Previously used code and data is often retained within the processor, in a *cache*, which is considerably faster to access than the system memory. Whether coming from memory or a cache, instructions are processed by the instruction decoder, which typically splits instructions into three or more smaller micro-operations (μops):

- Load: This stage fetches necessary data into the processor's fastest internal storage, the *registers*. All arithmetic is performed within registers. PC processors contain a number of *hidden registers*, which are used to temporarily hold data, when an instruction specifies a memory location as one of its parameters (operands).
- Execute: This stage performs the operation requested by the original instruction (e.g. a multiplication), on the values in the registers that were populated during the "load" stage.
- Store: The result of the operation is stored into a destination typically specified either explicitly by the instruction or implicitly, by the specification by the instruction set specification.

A load, execute and store can typically execute simultaneously, however the execute state of one instruction depends on the data loaded by the load stage, so a single instruction typically requires more than one cycle to run fully. Different stages of different instructions may however run simultaneously. This is analogous to a production line, where different stages of the manufacture of some product unit may run simultaneously on separate units, although each stage depends on completion of the previous one.

Real processors typically have more operation types than the three listed above, however only these three will be considered, for the sake of simplicity. The same principles apply for pipelines with many more operation types.

4.1.2. The pipeline

The micro-operations (μops) are put into a queue, called the *pipeline*. As some μops can run simultaneously with others, the pipeline actually contains several queues, alongside each other. The pipeline contains several *stages*, each of which can contain several instructions of different types (e.g. load/execute/store). All of the instructions in one stage may be executed simultaneously, in one clock cycle. To understand how this works, consider the following x86 assembly code*, which adds two to some number†:

```
mov bx, [cx]        ; Load the memory value from position CX into register BX
mov [sp], 2         ; Store a value of two to memory location from register SP
mov ax, [sp]        ; Load the value pointed to by SP into register AX
add ax, bx          ; Add the value of the BX register to the AX register
mov [cx], ax        ; Store the value of the AX register to memory located at CX
```

In this "Intel syntax", each line specifies an operation, the destination and optionally, some sources. A mov instruction moves the value in the second operand to the destination specified by the first operand. Square brackets (e.g. [sp]) indicate a memory location, so the second instruction may be decoded as "Store a value of 2 to the memory location specified by the sp register". The third instruction then loads the value that was just stored to [sp], into the ax register. Decoding this into load-execute-store micro-operations gives the following:

Instruction (x86)	Load from	Execute	Store to
mov bx, [cx]	[cx] (memory)	n/a	bx (register)
mov [sp], 2	n/a ("2" coded in instruction)	n/a	[sp] (memory)
mov ax, [sp]	[sp] (memory)	n/a	ax (register)
add ax, bx	ax, bx (registers)	addition	ax (register)
mov [cx], ax	ax (register)	n/a	[cx] (memory)

After decoding, we attempt to squeeze these instructions into the pipeline. Assume that "loading" and "storing" a *register* operand is instantaneous and that several can occur at either the start or end of a cycle, since registers are stored within the processor and are thus readily accessible. An "execute" or "store to memory" will not

* Typecasts have been removed from all assembly examples, normally the pointers (bracketed expressions) would be prefixed by some width specifier, e.g. BYTE PTR.

† In a C-syntax: `*cx += 2;`

be processed in the same cycle as a "load" from *memory* that it depends on, as loading from *memory* will take at least one cycle*. Assume that an "execute" requires a whole cycle to run, so a "store to memory" must be processed in the cycle after the "execute" that produced the value to be stored. A "store to register", being instantaneous, may run instantaneously at the very end of a cycle – immediately after the "execute".

These rules result in the following queue in the pipeline. Pipeline instructions have been colour coded to group them by their original assembly instruction.

	#1	#2	#3	#4	#5	#6	#7
LOAD	[CX]		[SP]	AX, BX	AX		
EXEC				ADD			
STOR	BX	[SP]	AX	AX	[CX]		

Recalling that the original code simply adds two to some number located at [cx] in memory, this resulting pipeline looks very inefficient: whilst we only have one "execute" cycle (which is good), we have loads in four cycles and stores in five cycles. Inspecting the pipeline further, the third cycle cannot execute before the second cycle as it depends on the value that is stored to [sp]. This value is already available though (the 2 in the code), so the delay is unnecessary. Additionally, cycle #4 depends on cycle #3, which loads this already-known value into a register. A simple optimisation for this code would be to reduce how much we move this known value around!

```
mov bx, [cx]    ; Load value from memory location specified by CX, into BX register
add bx, 2       ; Add value of 2 to BX
mov [cx], bx    ; Store value of BX to memory at CX
```

This now gives the following pipeline (execute columns, from left to right):

	#1	#2	#3	#4	#5	#6	#7
LOAD	[CX]	BX	BX				
EXEC		ADD					
STOR	BX	BX	[CX]				

While our optimisations have reduced the program to three cycles, each instruction occupies one whole cycle, so the advantage of using a pipeline is not apparent. Now consider the following code and its corresponding pipeline:

```
add [cx], 2
```

	#1	#2	#3	#4	#5	#6	#7
LOAD	[CX]						
EXEC		ADD					
STOR			[CX]				

* Loading from memory can take over 200 cycles even in modern processors, since several internal caches are searched before the processor resorts to communicating with the external memory banks.

The code now does not require a "store" in its first cycle, or a "load" in its final cycle. When this instruction follows some other instruction, then the lack of an initial "store" or final "load" becomes a useful property, as data for the next "execute" instruction can be loaded while the current "execute" instruction is being processed:

```
add [cx], 2
add [dx], 4
add [ax], 8
```

	#1	#2	#3	#4	#5	#6	#7
LOAD	[CX]	[DX]	[AX]				
EXEC		ADD	ADD	ADD			
STOR			[CX]	[DX]	[AX]		

While one add-an-immediate-value-to-memory instruction might require three cycles, two such instructions in succession only require four cycles. Similarly, four such instructions require only six cycles – hence with pipelining, four such instructions only take twice the number of clock cycles to execute completely than one instruction did, in this simplified model of a processor.

Realistically, not all "load", "store" or "execute" instructions can run in just one cycle. An integer addition may require an additional cycle to perform checks* on the result. Likewise, loading and storing from memory will require more than one clock cycle. Therefore, the simple seven-stage pipeline used in previous examples may "stall", waiting for some prolonged micro-operation to complete before being able to process subsequent operations.

Whilst most modern compilers (and indeed processors) will re-order instructions in order to maximise the efficiency of the pipeline (avoid empty slots, reduce stalling), being aware of the pipeline allows the programmer to write code in such a way as to make it easier for the compiler to optimise code for the pipeline. In C, this can be as simple as using array notation wherever possible (instead of pointers), which often has little or no adverse effect on the readability of the code, or the ease of future maintenance.[48]

4.1.3. Execution units

To negate the effect of "slow" instructions, a processor will often have several separate "execution units". Each unit can process a variety of operations, and all units

* Overflow: Is the result too small/large to be stored in the destination? Sign: Positive or negative? Carry/borrow, zero, parity...

may execute simultaneously. The set of operations that a particular execution unit can process will typically differ between execution units in the same processor. The execution units of a modern processor based on the *Sandy Bridge* microarchitecture are shown in Figure 4-A. A more detailed diagram of this architecture may be found in source #49.

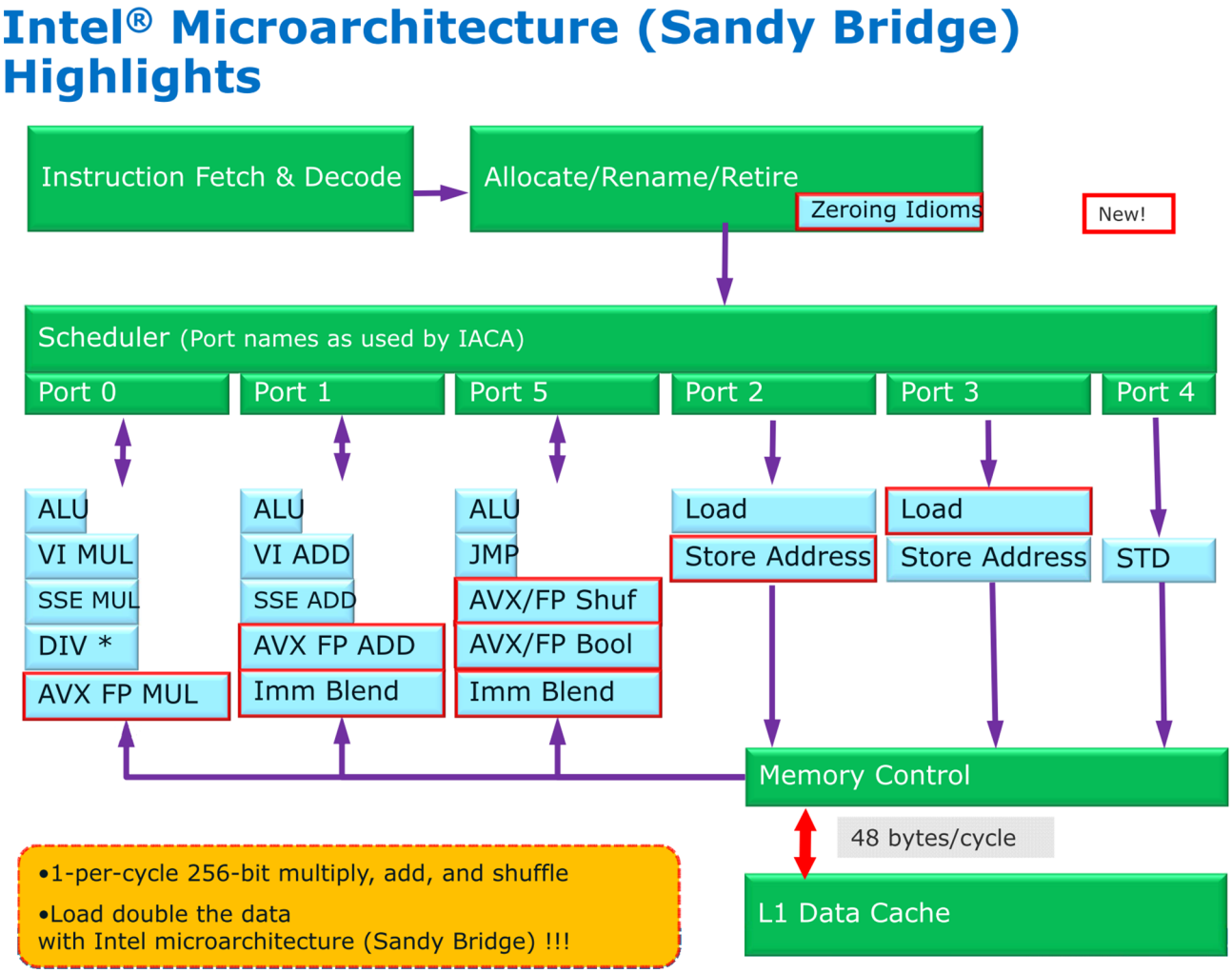

Figure 4-A: Block diagram of execution units in the Intel Sandy Bridge microarchitecture. [50,52]

The SSE and AVX vector instructions are split across three execution units, and the arithmetic logic units (ALUs) are duplicated – so an addition, multiplication and shuffle can all execute simultaneously provided they do not depend on each other, or three integer operations could execute simultaneously. This reduces the amount of time that a processor spends stalled, as several arithmetic operations may execute simultaneously, while the results of the previous instructions are being stored, or the parameters for the next instructions are being loaded. Of particular scientific interest (for large dot-products and matrix multiplications) is that an SSE multiplication may execute alongside an SSE addition and an integer addition. Analyse the dot-product code in section 4.3 to see why this particular combination is useful.

4.2. Data types

There are many ways to represent numbers on paper, for example:

- Hindu-Arabic numeral system: Perhaps the most common system, using a series of digits from zero to nine with positional notation. For example: 42, 0.03, 2501.
- Tally[*]: A series of strokes and/or dots are drawn, with a quantity to match the number being represented. For ease of reading, these are commonly grouped in some way, for example: 𝍸||, ⧅┌
- Scientific notation: An extension of the Hindu-Arabic system, where some exponential scale factor is introduced, for example: 1.6×10^{-19}, 3×10^{8}.

There are also many ways to represent numbers electronically. The decimal (base-10) number system is typically inconvenient for fast electronics, so the binary (base-2) system or less commonly, the ternary[†] (base-3) system is used. As with paper methods, these numbers may be written as a single series of digits, or analogously to scientific notation (i.e. with some exponent). The way that a number is represented affects the range of values that can be stored, the precision of the stored values (and calculations involving them) and the speed of arithmetic operations on those values.

4.2.1. Integer types

Programming in native languages[‡] requires some knowledge of machine *data-types*. Computer arithmetic is fastest when working with whole numbers, electronically stored in a similar way as to how whole numbers are written down (a series of digits). Types which store whole numbers are typically referred to as *Integer* types in most programming languages. Just as larger decimal numbers require more digits to be written down on paper, a size of integer type must be chosen which can hold the numbers in use. Common PCs work with integers comprising 32 or 64 binary digits ("bits"). As each digits has two possible values (zero or one), this gives a maximum number of possible values of 2^{32} and 2^{64} respectively. All the bits could be used to store the number's digits ("unsigned integer"), resulting in a range from

[*] Sometimes incorrectly referred to as base-1

[†] The ternary system is mainly an academic curiosity.

[‡] And some interpreted languages – e.g. Java, C#, VB

zero to $2^{32}-1 \approx 4\text{bn}$ for a 32-bit number. Alternatively, one digit could be used to store the *sign* of the integer, i.e. is it positive or negative? This halves the maximum value that the data type can store, but allows negative values to be stored. The number of *bits* used to store a data type is often referred to as the *width* of the data type.

Note that integer types are *precise*: if a certain integer type has a range of -100 to $+100$, then every integer value within that range can be represented uniquely by the data type. Consequently, the result of multiplication, addition and subtraction of any numbers of that data type can also be represented precisely by the same type, provided the result does not exceed the range of the type. Division of integer types typically produces a result rounded to an integer; the direction of the rounding depends on the compiler and/or processor used.

Some common integer types are listed below. Note that the C standards specify a *minimum width*, for integer types – some implementations may use greater widths than are specified here.

Width/signed	Minimum	Maximum	C/Pascal name
Unsigned			
8-bit unsigned	0	255	unsigned char/Byte
16-bit unsigned	0	65,535	unsigned short /Word
32-bit unsigned	0	4,294,967,295	unsigned long/Cardinal
Signed			
16-bit signed	-32,768	32,767	signed short/SmallInt
32-bit signed	-2,147,483,648	2,147,483,647	signed long/LongInt
64-bit signed	-9.2×10^{18}	9.2×10^{18}	signed long long/Int64

Table 4.a: Common integer data types, available on all modern PCs.

The Javascript example may be coded in Pascal, using integer types.

```
program TwoPlusTwo;
var
   a, b, c: LongInt;
begin
  a := 2;
  b := 2;
  c := a + b;
end.
```

Or alternatively, in C:

```
int main(int argc, char *argv) {
   int a, b, c;
   a = 2;
   b = 2;
   c = a + b;
}
```

Note that like the JavaScript implementation, this simply calculates two plus two, then store the result to memory or a register (determined by the compiler). When the program finishes (and closes), the result is destroyed. This program has no practical

purpose; it is simply used to demonstrate the apparent similarity between different programming languages, and usage of data types.

4.2.2. Fixed-point and rational types

While integer types are fast and precise, the perfect precision limits the range of values that may be stored within such a type. For example, in common 32-bit integer types encountered on PCs, values of ten billion or one quarter could not be represented. Ten billion is above the range of the unsigned 32-bit integer (specifically, four billion) and one quarter is not a whole number. There are various solutions to this problem, all balancing precision, speed and size. For storage of fractions, one may combine two integers to produce a *rational* number. One (usually signed) integer stores the numerator of the fraction, while the other (commonly unsigned) stores the denominator. Multiplication, division, addition and subtraction may be achieved using the same algorithm that is taught in schools for working with vulgar fractions on paper. As an operation on two rational types involves working with four integers, such operations will be slower than working on just two integers, however the ability to work with non-integer values will outweigh this drawback in some situations.

A fixed value for the denominators of all rational types could be implicitly assumed, removing the need for it to be stored with each number. For example, when adding shop prices* in British Pounds Sterling or Euros, an implicit denominator of 100 could be assumed. Rational types (e.g. the Pascal *Currency* type) with implicit denominators are called *fixed-point* types, and are faster and more compact than rational types, but are slower than integer types.

4.2.3. Floating-point types

Rational types and fixed-point types allow handling of fractions, however they do not overcome the range limitation of integer types. Often in scientific computation, perfect precision is not necessary; hence, some precision may be traded for increased range. For example, the speed of light in a vacuum is defined exactly as $c_0 = 299,792,458\ \mathrm{m/s}$. The Coulomb constant is defined as $k = c_0^2 \times 10^{-7}\ \mathrm{Nm^2/C^2}$. While this value is known exactly (with 17 decimal digits), it is often sufficient in

* i.e. smallest unit of currency is one pence, 1/100 of a pound.

calculations to round it to a few places, as physical measurements will usually not be precise enough for the rounding to have a significant effect on the result. The rounded constant may be written in scientific notation as 8.99×10^{9}. A similar notation is used by computers, for inexact representation of very large and very small numbers. Scientific notation is composed of a *mantissa* multiplied by a *base* that is raised to an *exponent* as shown in equation 3.3. Conventionally, the mantissa has one and only one non-zero digit immediately to the left of the radix, and any number of digits to the right of the radix, illustrated in equation 3.4.

$$\text{mantissa}\times\text{base}^{\text{exponent}} \qquad \text{e.g: } c_0 = 2.99792458\times10^{8} \qquad 3.3$$

$$\text{mantissa} = d_0 \bullet d_1 d_2 d_3 d_4 d_5 ... \qquad \text{e.g: } \pi = 3{\cdot}14159... \qquad 3.4$$

In floating-point formats, the base is implicitly either two or ten (depending on the data type) and the leading digit of the mantissa (for base two) is often implicitly set to one. With these (and other) assumptions, the amount of information that must be stored to describe a number is reduced, boosting either the range or precision available within a certain width. The result is that an IEEE 754 32-bit floating-point type can store values as small as 10^{-38}, as large as 10^{+38}, with a precision of approximately seven decimal figures. Arithmetic on floating-point types is slower than for integer types, but the algorithms for floating-point arithmetic are embedded in the circuitry of modern processors. This yields a substantial performance increase to such operations compared to software-based implementations. There are several floating-point types, differing by the number of bits allocated for the mantissa and the exponent, or by the way that the data is arranged within the type. Below are two types, defined in IEEE 754-1985[51,*], in addition to a third 80-bit type available on most PC processors:

Width/description	Min/max magnitude	Precision	C/Pascal name
32-bit single-precision	1.5×10^{-45} / $3.4\times10^{+38}$	~7 digits	float/Single
64-bit double-precision	5.0×10^{-324} / $1.7\times10^{+308}$	~15 digits	double/Double
80-bit extended-precision	3.4×10^{-4932} / 1.1×10^{4932}	~19 digits	(disputed[†])/Extended

Table 4.b: Common floating-point types, available on all modern PCs.

[*] Since superseded by 754-2008, which defines more types

[†] Some C compilers accept *long double* to specify an 80-bit type, other C compilers refuse to support Extended precision, and map it to double-precision. The specifications for the basic data types in the C standards are defined very weakly.

The example algorithm, coded to use "single-precision" floating-point types looks remarkably similar to the integer version in either C or Pascal:

```
program TwoPlusTwo;
var
   a, b, c: Single;
begin
  a := 2;
  b := 2;
  c := a + b;
end.
```

Equivalent program, in C:

```
int main(int argc, char *argv) {
   float a, b, c;
   a = 2;
   b = 2;
   c = a + b;
   return 0;
}
```

4.3. Vectorising

Often in scientific and multimedia applications, a single operation will be applied to many sets of floating-point values. Examples of this include vector addition [equation 3.5], matrix multiplication [equation 3.6] and the dot product [equation 3.7].

$$\vec{a} = (a_i)_i ;\ \vec{b} = (b_i)_i ;\ \vec{c} = \vec{a} + \vec{b} = (a_i + b_i)_i \qquad 3.5$$

$$\mathbf{A} = (a_{ij})_{ij} ;\ \mathbf{B} = (b_{ij})_{ij} ;\ \mathbf{C} = \mathbf{AB} = \left(\sum_k a_{ik} b_{kj} \right)_{ij} \qquad 3.6$$

$$\vec{a} = (a_i)_i ;\ \vec{b} = (b_i)_i ;\ C = \vec{a} \cdot \vec{b} = \sum_i a_i b_i \qquad 3.7$$

A historic supercomputer, the ILLIAC IV, attempted to accelerate processes like these by having several arithmetic logic units (ALUs), each operating on separate data, but all driven by the same instructions at once – nowadays referred to as "SIMD" (single-instruction, multiple-data). Hence, a vector addition could be performed in a similar amount of time as an individual floating-point multiplication since each pair of elements was added simultaneously on different ALUs. 21st century PC processors can typically operate on four pairs of single-precision data at once, using SIMD Streaming Extensions (SSE). Recent instruction sets such as AVX allow even larger amounts of data to be processed in one instruction, while FMA allows multiplication and addition to be combined into one instruction. Vectorisation technology is continually evolving as processors become more advanced; a timeline of SSE instruction sets as implemented in Intel processors is shown below:

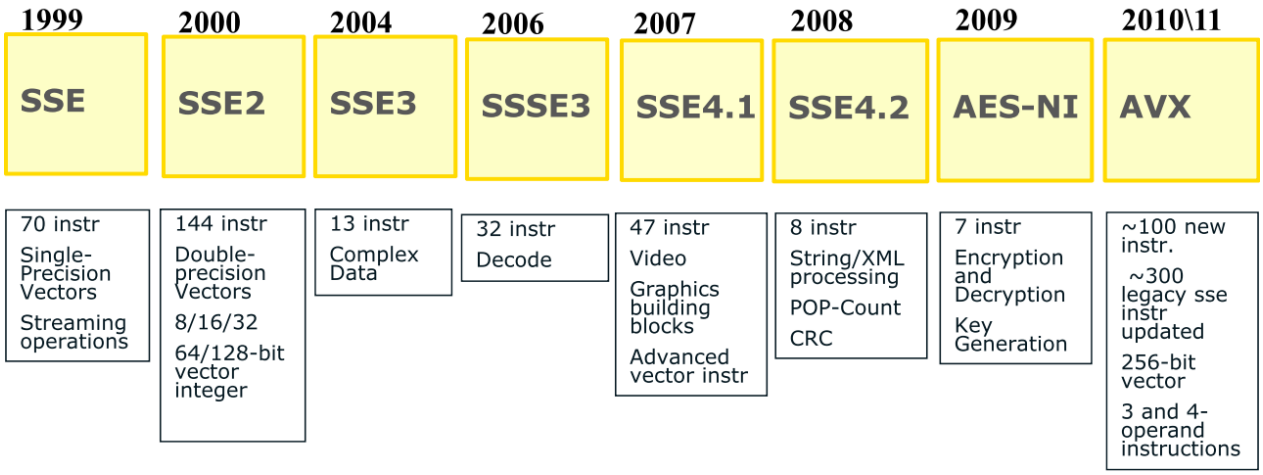

Figure 4-B: Timeline of SIMD Streaming Extensions (SSE) in Intel processors.[52]

Typically, there is no convenient way to instruct the compiler to use these technologies in the code; one must either rely on the compiler's analysis of the code (assuming the chosen compiler can automatically vectorise code) or one must manually specify the low-level instructions to use. The pure C code to perform a dot product is shown below. Modern compilers would usually be able to detect that this code can be vectorised, and would vectorise it automatically.

```
float dot(float *a, float *b, int length) {
   float result = 0;
   int i;
   for (i = 0; i < length; i++)
      result += a[i] * b[i];
   return result;
}
```

Although modern compilers would usually be able to vectorise this routine automatically, routines that are more complex may require manual optimisation. An example of such an optimisation for the previous dot-product code is shown below*, using SSE "intrinsics", which represent machine-level SSE instructions with a C syntax.

* This version requires that the length of the vectors is a multiple of four, or that the vectors are zero-padded to a length that is a multiple of four

```
#include <intrin.h>
#include <xmmintrin.h>

float dot(float *a, float *b, int length) {
   float result;
   int i;
   __m128 total, partialsum = _mm_setzero_ps();
   for (i = 0; i < length; i += 4)
      partialsum = _mm_add_ps(partialsum,
         _mm_mul_ps(_mm_load_ps(&a[i]), _mm_load_ps(&b[i])));
   total = _mm_hadd_ps(partialsum, partialsum);
   total = _mm_hadd_ps(total, total);
   _mm_store_ss(&result, total);
   return result;
}
```

As more optimisations are performed, the code becomes less readable. Whist optimisations may reduce the time required for a program to run, the increased amount of time required to develop the program and to test it may negate the benefits of optimisation. Code that is not expected to need modifying for a long time may be optimised heavily; the optimisation and testing will take a considerable amount of time, but the resulting program will run very quickly. For code that is likely to change frequently, or to only be run a small number of times, the amount of optimisation that should be performed on any piece of code is a delicate balance between production/testing time and run time. An example of somewhat illegible, but optimised code is shown below. It is an optimised version of the dot-product C routine.

```
Parameter locations:
   float *a          in  eax
   float *b          in  edx
   int length        in  ecx
   float result      out ST(0)

Code:
   xorps xmm0, xmm0                    ; zero the accumulator
   test ecx, ecx                       ; sanity check:
   jle short @EndLoop                  ;   is vector length greater than zero?
   lea edx, [edx-eax]
   shr ecx, 2                          ; fast way to divide integer by four
   @BeginLoop:                         ; "for"-loop
      movaps xmm1, XMMWORD PTR [edx+eax]; load data from one vector
      mulaps xmm1, XMMWORD PTR [eax]   ; entrywise product with other vector
      add eax, 16                      ; move data pointer onto next block
      dec ecx                          ; decrease loop counter
      addps xmm0, xmm1                 ; accumulate entrywise products
      jnz short @BeginLoop             ; do next iteration of "for"-loop
   @EndLoop:                           ; end of "for"-loop
   haddps xmm0, xmm0                   ; these horizontal additions sum the four
   haddps xmm0, xmm0                   ;   partial sums of the accumulator
   movss DWORD PTR [esp-4], xmm0       ; these two transfer the answer to the
   fld DWORD PTR [esp-4]               ;   FPU. Not terribly efficient...
   ret                                 ; end of subroutine (return to caller)
```

If in future, single-precision floating-point was insufficient, i.e. a larger range or more precision was required, then the original pure C code could be quickly modified

to use the *double* type instead of *float*. The SSE-optimised C and the above assembly code however would require considerably more changes. Scientific software typically does more than just calculate dot-products, but the balance between execution time and development time demonstrated with this simple code is just as applicable to much larger and more complex scientific programs.

The most recent PC vectorisation instruction sets, FMA and AVX, could have also been used, however the code would not be able to run on an older or simpler processor that did not support such instructions. Lab computers are typically upgraded/replaced infrequently; so cutting-edge instruction sets should be avoided for software intended for lab computers, unless a suitable computer is to be supplied with the software. For this reason, such instruction sets will not be considered.

4.4. Multithreading

The maximum rate at which a processor can execute instructions is typically determined by two main factors[*]:

- Clock speed – the rate at which electronic clock signals trigger the computational circuitry within the processor. Typically, the entire computer is driven by some *base clock*, often at hundreds of megahertz. The processor may execute several instructions in each base clock cycle, the maximum number determined by a *multiplier ratio*. The processor's clock speed is the product of the base clock rate and the multiplier, for example a system with a 100 MHz base clock speed and a processor multiplier rate of 42 will have a processor clock speed of 4200 MHz (4.2 GHz).
- Core[†] count – Modern processors contain several "cores", each capable of simultaneously executing different code and working with different data.

Just as code can be modified (vectorised) to take advantage of vector units within the processor, and instructions can be combined and re-ordered in order to exploit the pipelined nature of modern processors, algorithms can also be *multithreaded* in order to use multiple cores simultaneously. Multithreading comes under the umbrella of

[*] There are other factors, for example the memory bandwidth and the cache latency.

[†] Note that AMD's marketing uses "core" to refer to what IBM and Intel call a "thread", and uses "module" to refer to what is commonly accepted within the industry to be a "core".

parallelisation, which involves splitting a single task into chunks that can be executed simultaneously and near-independently. Each separate execution path (in the context of multithreading) is referred to as a *thread*. Each core has its own individual instruction decoder(s), pipeline(s), execution units, registers and cache.

In the case of a large dot-product, the first half of each vector could be "dotted" in one thread, while the second halves are simultaneously "dotted" in another. These partial dot-products could then be collected by one thread and summed to give the final answer. The stages of this multithreaded dot-product are:

- Dispatch: The data is split into separate chunks, which are dispatched to each worker thread. As typical PCs have one node of memory, shared between all cores, this step consists only of deciding *what* data is to be processed in each thread. More advanced systems, using NUMA (none-uniform memory architecture) would generally require data to be moved between memory nodes at this stage.
- Execution: The partial dot-product routines on each core are started. Each thread can begin this immediately after the dispatch. Some other operations (where threads communicate during execution) require the dispatch to complete in all threads before execution can begin.
- Cleanup: The partial results from each thread are collected into one thread and summed together. For some operations, all threads must finish execution before this stage can occur. For a parallel dot-product, each thread can add its partial result to some shared running total as they complete, provided some mechanism is in place to ensure that no two threads attempt to do this simultaneously (e.g. atomic operations, critical sections).

Other operations (e.g. matrix inversion) may require more inter-thread communication. Communication and co-ordination between separate cores is beyond the scope of this report*; it is sufficient to say that it often has a negative effect on the over all performance of the code. Hence, multithreaded routines must usually minimize the amount of synchronisation and co-ordination performed between individual threads.

* Involving synchronisation objects, LOCK prefixing and shared memory.

Parallelisation in C has been greatly simplified by technologies such as the following:

- MPI – Message Passing Interface: this provides a simple and robust way for separate instances of a program (or of different programs) to communicate, whether they are running on different cores on the same computer, or running on many separate computers within a grid computing network such as IBM's Blue Gene project and the numerous Cray supercomputers around the world.
- OpenCL/CUDA/OpenACC – These provide relatively easy ways to tap into the computational power of graphical processing units (GPUs/GPGPUs), which may be seen as extreme vector processors capable of running one instruction on dozens of pairs data simultaneously.
- OpenMP – This provides incredibly simple ways to multithread algorithms, with minimal changes required to the algorithms. The code often remains readable and maintainable.

OpenMP and OpenACC can provide significant performance improvements, with minimal extra time required to implement. OpenACC currently requires specialist hardware, so will not be considered here. OpenMP is mature and is fully implemented on many major operating systems and system architectures.

The pure-C dot-product routine is shown again below (assume that the compiler can automatically vectorise the main loop):

```
float dot(float *a, float *b, int length) {
   float result = 0;
   int i;
   for (i = 0; i < length; i++)
      result += a[i] * b[i];
   return result;
}
```

An OpenMP–multithreaded version of this routine is shown below:

```
float dot(float *a, float *b, int length) {
   float result = 0;
   int i;
   #pragma omp parallel for reduction(+:result)
   for (i = 0; i < length; i++)
      result += a[i] * b[i];
   return result;
}
```

For multithreading without OpenMP, several calls to operating-system specific routines would be necessary in order to start ("fork") the extra threads, a separate subroutine containing the threaded code to execute in each thread would have to be

created, and synchronisation code would be necessary in order to detect when all threads have finished working. With OpenMP, the code that manages the threading is wrapped in simple directives ("pragmas"). Multithreading an algorithm can sometimes be as simple as adding *one* line of code to the original single-threaded routine, as shown above.

4.5. Benchmarking

Parallel and non-parallel versions of the vectorised and non-vectorised implementations were benchmarked on a 4.83 GHz Sandy Bridge quad-core processor, their speeds are compared in Figure 4-C.

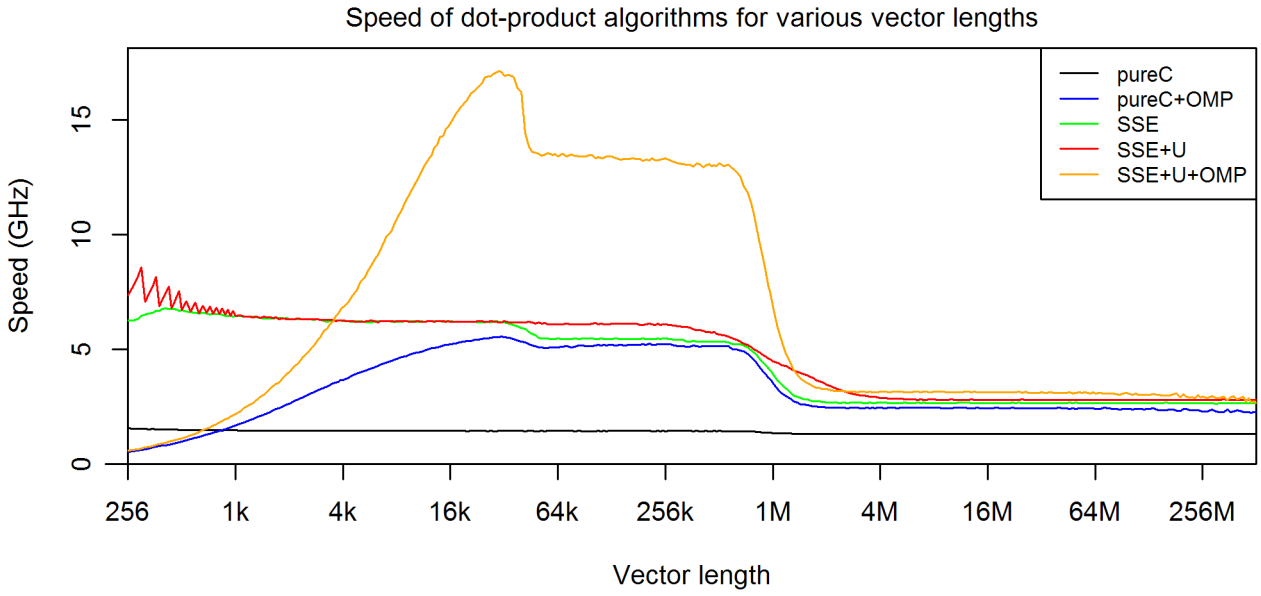

Figure 4-C: Comparison of the performance of various dot-product algorithms, with varying vector sizes. "U" = loop unrolling optimisation, "SSE" = SSE4 vectorisation optimisation, "OMP" = OpenMP multithreading optimisation.

The largest cache on the processor (level-3) is eight megabytes in size. The peak bandwidth of this cache is approximately 132 GB/s on the processor used for this benchmark, in contrast to the peak system memory bandwidth in the configuration used, which was 30 GB/s. The processor has a 64-bit (8-byte) *word size*. This is the width of the data paths and registers. This allows the processor to store integers and memory addresses with widths of up to 64-bits, allowing it to efficiently perform arithmetic on integers of 64-bit width and smaller. Caches and memory can typically transfer blocks of two, four or eight words between each other. Each core of the Sandy Bridge processor can load six such words (48-bytes) per cycle. The level-3 (last level) cache on the system used was 8 MB in size and shared between all four cores. Each

core has its own level-2 cache, which is 256 kB in size, giving a total level-2 cache size of 1 MB. The performance graph of Figure 4-C is shown in Figure 4-D, but the axes are now in terms of the dataset size and data bandwidth. Notice that for all algorithms, the performance drops to some asymptotic values, as the data size exceeds the level-3 cache size.

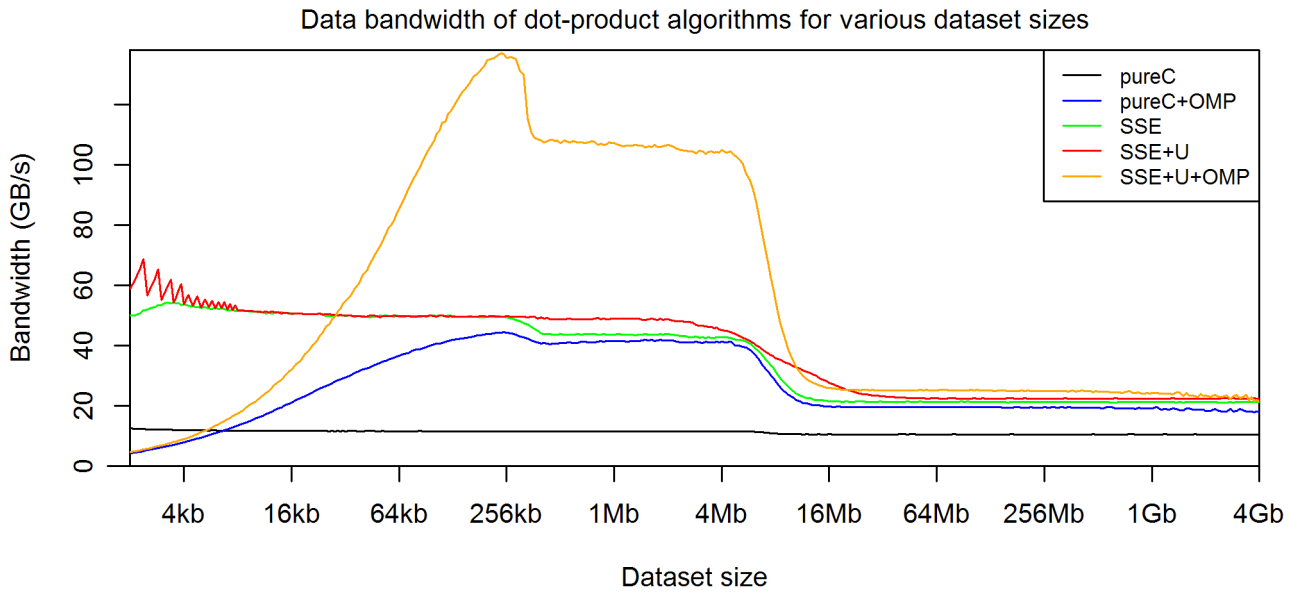

Figure 4-D: Performance of various dot-product algorithms, plotted against the dataset size

The speed of the single-threaded, non-vectorised algorithm (black: pureC) is primarily limited by the rate at which the processor can calculate multiplications and additions, hence does not change until the system memory is required – while the memory accesses don't bottleneck the algorithm, they do introduce some latency, which reduces the performance slightly. The SSE and unrolled SSE algorithms (green, red) achieve approximately four times the speed of the C algorithm when not limited by memory bandwidth. Even when the data bandwidth becomes a bottleneck, they achieve double the performance of the C algorithm. This is most likely due to the C algorithm fetching eight bytes for each cycle of the dot product (a pair of 32-bit floats), whereas the SSE versions fetches thirty-two bytes at a time (a pair of vectors, each containing four 32-bit floats). The C version loads two 64-bit words per iteration but then discards half of each word, whilst the SSE version loads four words per iteration, but uses all the information in them. The C version "wastes" half of the data that it loads, halving its performance when the rate of data access (i.e. memory bandwidth) becomes a bottleneck.

Compared to the plain SSE version (green), the "unrolling" optimisation (red*) results in a performance increase that ranges from negligible to approximately 15%. Interestingly, this algorithm does not experience the performance drop that the non-"unrolled" version does as the dataset size exceeds 256 kB. It is thought that this drop has some relationship to the level-2 cache (which is 256 kB in size, per core) and that the absence of a drop for the unrolled version is due to the prefetching optimisation, which instructs the core to load data several iterations before the data is processed. This implies that the bandwidth of the caches is not a significant constraint on the performance of the algorithm, but that the higher latency of the level-3 cache will cause a significant drop in performance if data is not pre-fetched in advance.

For dataset sizes less than 32 kB, the overhead of thread management and synchronisation results in a decreased performance for the multithreaded SSE algorithm. Beyond this size, the performance of the algorithm increases, as the time incurred by the thread management becomes less than the time saved by calculating in parallel. This increase abruptly stops as the dataset size approaches (and exceeds) the 8 MB size of the level-3 cache, as the system memory bandwidth becomes the bottleneck (instead of calculation speed).

The vectorised algorithms are always faster than the non-vectorised (scalar) versions, as the vectorised versions perform more arithmetic per processor cycle. Unrolling gives a slight increase in performance across the entire vector length range that was used. Multithreading is detrimental to performance for algorithms that run for short amounts of time (comparable to the time of the threading overhead), but larger vector sizes gain a significant (~2-3x) performance increase with multithreading on this particular system. Multithreading also gives a slight performance increase for bandwidth-limited execution, as more cache is available for use. Matrix multiplication may be viewed as a series of dot-products, so this conclusion also applies to matrix multiplications, where the width of the left matrix (alternatively, the height of the right matrix) is the vector size for each individual dot-product operation. The optimal algorithm to use for any operation (not only dot-products) is determined by the size

* A "prefetching" optimisation was also used for the unrolled versions. The loops were unrolled by a factor of sixteen.

and structure of the input data, in addition to the hardware available. A graph of the speeds of the algorithms, measured relative to the speed of the pureC implementation is shown in Figure 4-E, below.

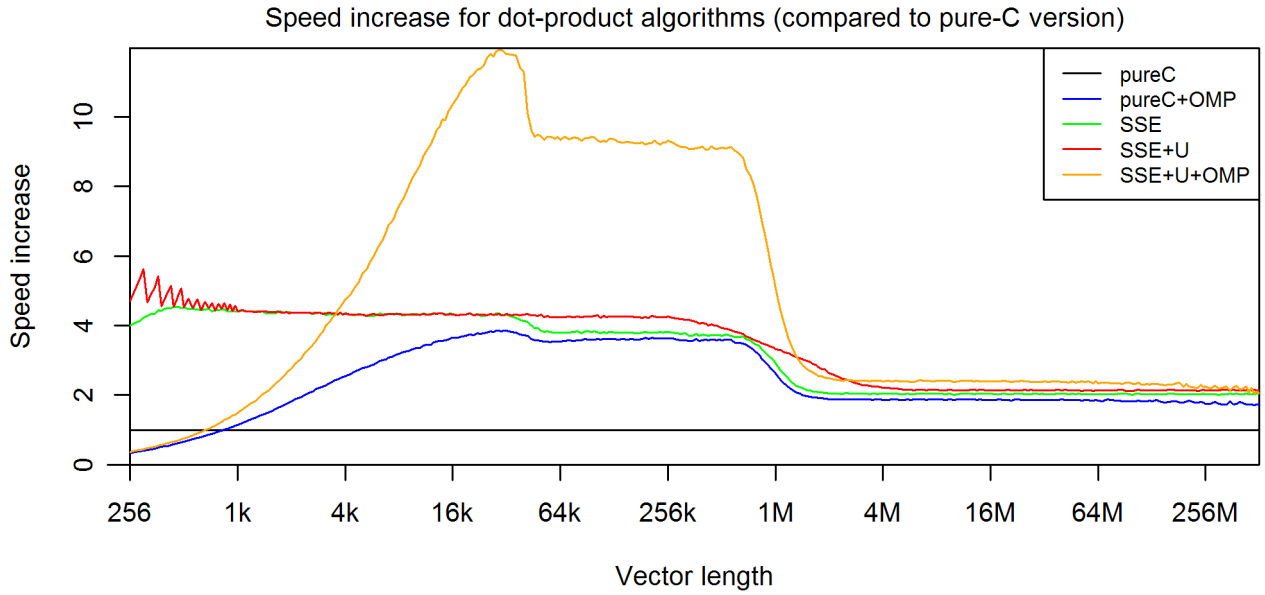

Figure 4-E: Performance of various dot product algorithms, relative to the pureC implementation.

The purpose of this benchmarking example is to demonstrate the significant difference in performance between several algorithms designed to perform identical tasks, and to emphasise the importance of optimisation. Dot-products of large vectors may be calculated even more rapidly on GPGPU hardware than by use of the previous algorithms on CPUs. Whilst nonlinear solvers and other common scientific computation tasks are considerably more complex than dot-products, the benefits of optimisation apply similarly to the more advanced algorithms.