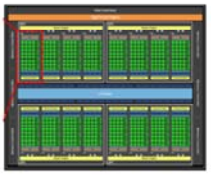
Processor Architecture

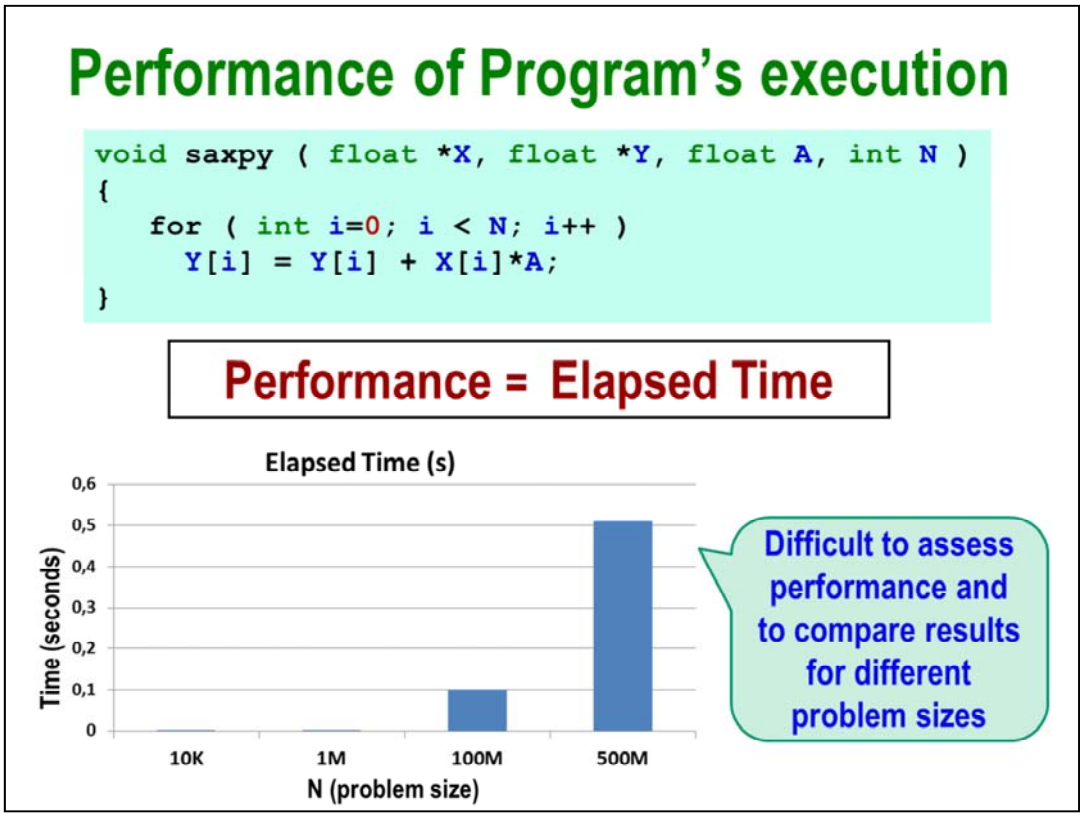


Processor Details to understand Performance

Instruction Set Architecture (ISA)

SIMD (Single Instruction Multiple data) operation and Code Vectorization

Performance analysis requires relating results with the machine instructions that the computer ultimately executes. The expectation is that a good analysis of the machine code will straightforwardly lead us to a certain types of code optimizations that will allow us to achieve better performance outcomes.



Performance is measured as "*Elapsed Time*" or "*Wall-clock time*", which means the time elapsed since the first instruction of the program starts running on the processor until the last instruction has just been executed.

The code example is called the saxpy function and is often used as a *benchmark* to measure and compare computer performance. The following website provides more information: https://devblogs.nvidia.com/parallelforall/six-ways-saxpy/ :

SAXPY stands for "Single-Precision A·X Plus Y". It is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. SAXPY is a combination of scalar multiplication and vector addition, and it's very simple: it takes as input two vectors of 32-bit floats X and Y with N elements each, and a scalar value A. It multiplies each element X[i] by A and adds the result to Y[i].

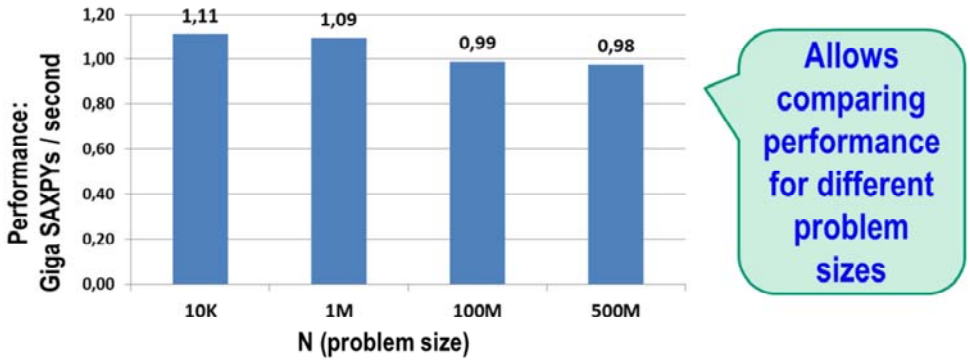
The chart depicts elapsed time as a function of problem size, where problem size is determined by N, the size of the input vectors. The decision of picking a bar chart is correct, since the problem sizes instances selected do not follow a simple distribution (linear, quadratic or exponential). However, depicting time is not a good decision, since time values are too different.

Next we will show an alternative way to present performance.

Performance of Program's execution

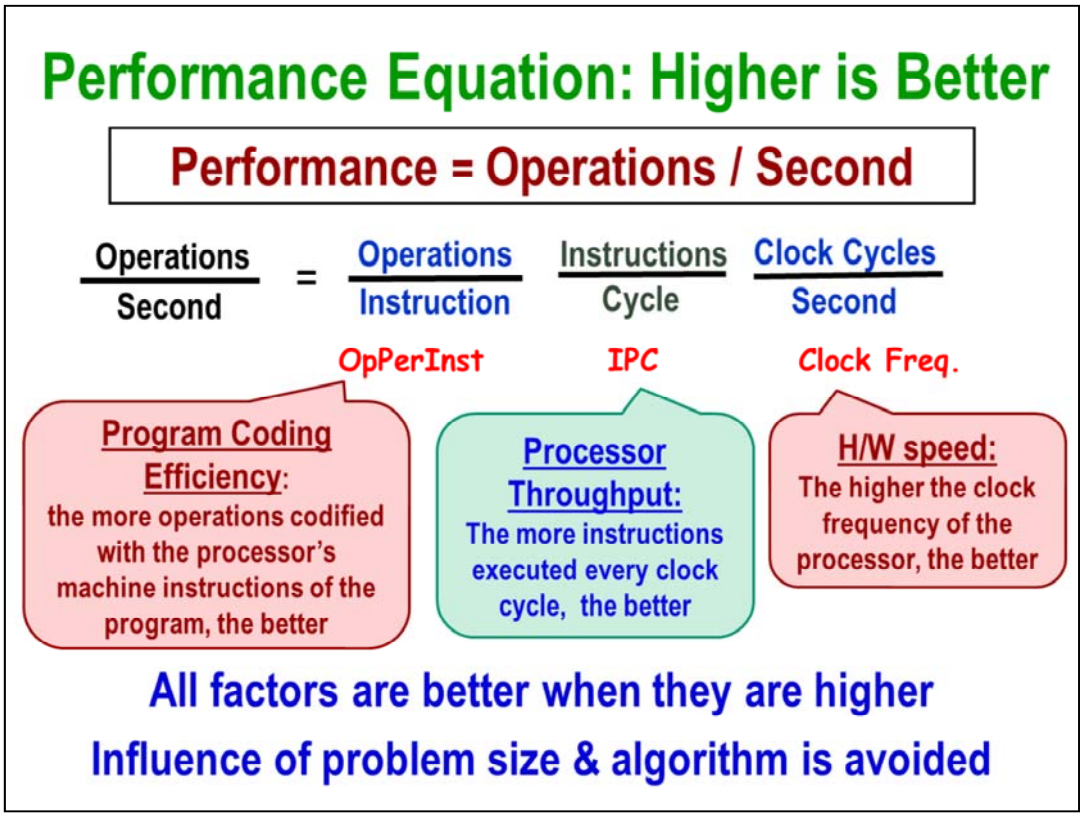
```
void saxpy ( float *X, float *Y, float A, int N )
{
   for ( int i=0; i < N; i++ )
     Y[i] = Y[i] + X[i]*A;
}
```

Performance = SAXPY operations / second



Sometimes it is a better idea to represent performance as a ratio of work done per unit of time. The problem is how to measure the work done by the program. We have to identify some general, high-level, and simple operation that can be used to measure work. In this example, we could use an operation called read-multiply-read-add-write, but it is more simple if we simply refer to saxpy operations. The advantage of this definition is that it is very easy to count the number of saxpy operations executed by the program: N.

The ratio of (Giga) saxpy operations per second is useful to compare the performance for different problem sizes. Here we see a small performance degradation (from 1.11 to 0.98) as the problem size is increased from 10K to 500 M. It is not a very severe problem, but later we will see an example that corroborates that this kind of chart can be a very helpful tool for a performance engineer.

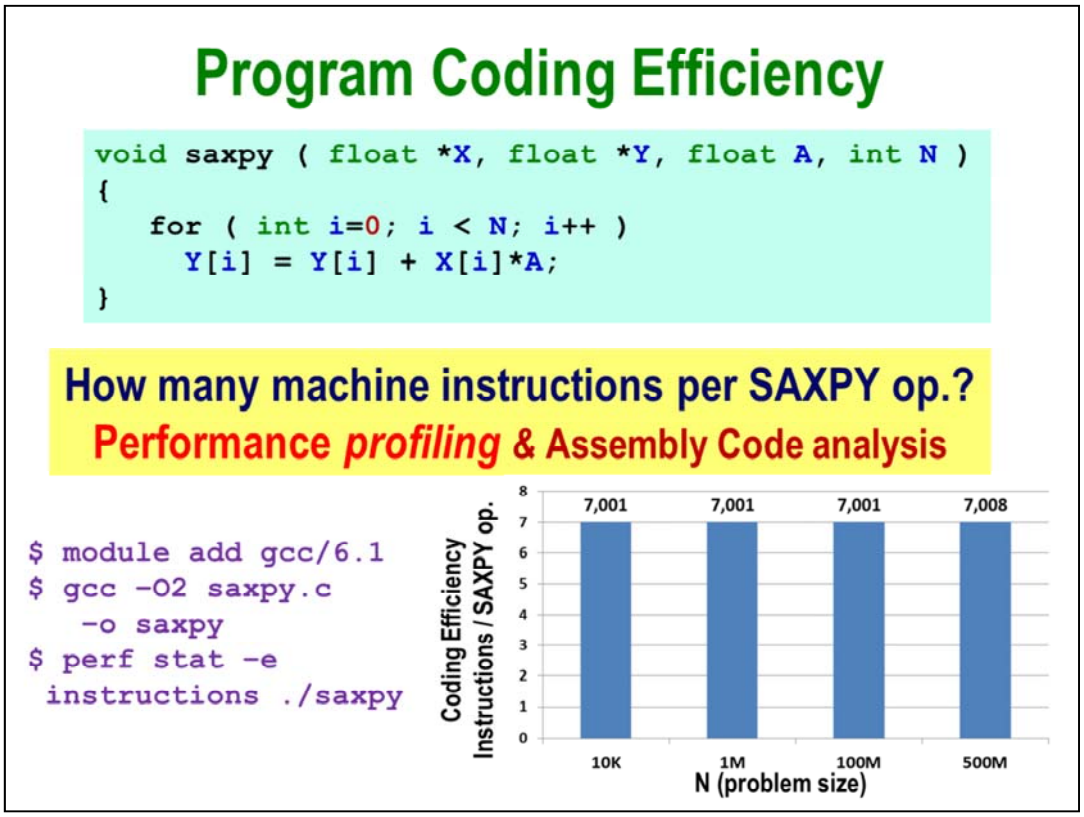


We normalize execution time and express performance as the total amount of work (operations) performed per second. This metric can be "explained" as the multiplication of three factors: (1) the program encoding efficiency (average number of operations that are codified per machine instruction, or per thousands of machine instructions); (2) the processor throughput (average number of machine instructions executed per clock cycle); and (3) the H/W speed, measured as clock cycles per second (or clock frequency).

Program encoding efficiency depends on how well the compiler translates the statements expressed in a high-level language, like C or Java or Matlab, into low-level machine instructions, the ones that the computer really understands and executes. Whenever we say instruction we mean machine instruction. Instructions perform simple operations (copy, addition, comparison, …) on data operands of basic types (32-bit integers, 32-bit real numbers in floating-point representation, 8-bit characters …). Changing the compiler or the processor means that a different codification is obtained, probably requiring more or less machine instructions.

Processor throughput depends on the internal organization (or microarchitecture) of the processor. Most processors today are able to execute multiple instructions simultaneously, giving an average rate of instructions executed per cycle (IPC) higher than one. Many events may prevent achieving maximum efficiency from the processor, like the occurrence of data dependences between instructions or long-latency memory operations. Those performance hazards are related to the actual program instructions, so at the end of the day the achieved IPC depends both on the program characteristics and on the processor microarchitecture.

Finally, clock frequency mostly depends on the technology used to implement a processor and the cleverness of the engineers designing the silicon gates and internal connections. High clock frequencies imply high energy consumptions (energy has approximately a cubic dependence on clock frequency) so that using lower clock frequencies is a way to save energy at the expense of lower compute performance.



A program can execute faster by executing less machine instructions. You can accomplish this by using a better algorithm (or computation strategy), or by codifying the algorithm more efficiently using a high-level programming language. The compiler is responsible for translating between high-level language sentences and low-level machine instructions. You can provide some information to help the compiler improve its work. You can also activate or deactivate compiler optimization strategies by using compiler flags on the invocation of the compilation process.

The perf command can be used to profile the execution of a program and to inspect the assembly code. The slide shows how to invoke the profiler to count the total number of machine instructions executed on behalf of the program. This number is used to measure the empirical program coding efficiency, that is to say, the average number of machine instructions executed for every saxpy operation. In this case this number is 7, but it is not easy to assess that value, and tell if it is a big (bad) number or a small (good) number.

To estimate how good or bad the value of coding efficiency of the program is, we have to know how the compiler works and what is the instruction set of the destination processor of the execution. The operation of the compiler is very complex and the set of instructions is usually extended with new generations of processors. We can only value this number to a fair measure if we take a look at the assembly code generated by the compiler and compare it with our expectations of what the proper implementation should be. That is, it is a task that requires extensive knowledge and an important degree of experience. But you can not learn anything without having been ignorant before: let's do it!

Program Coding Efficiency

```
void saxpy ( float *X, float *Y, float A, int N )
{
   for ( int i=0; i < N; i++ )
     Y[i] = Y[i] + X[i]*A;
}
```

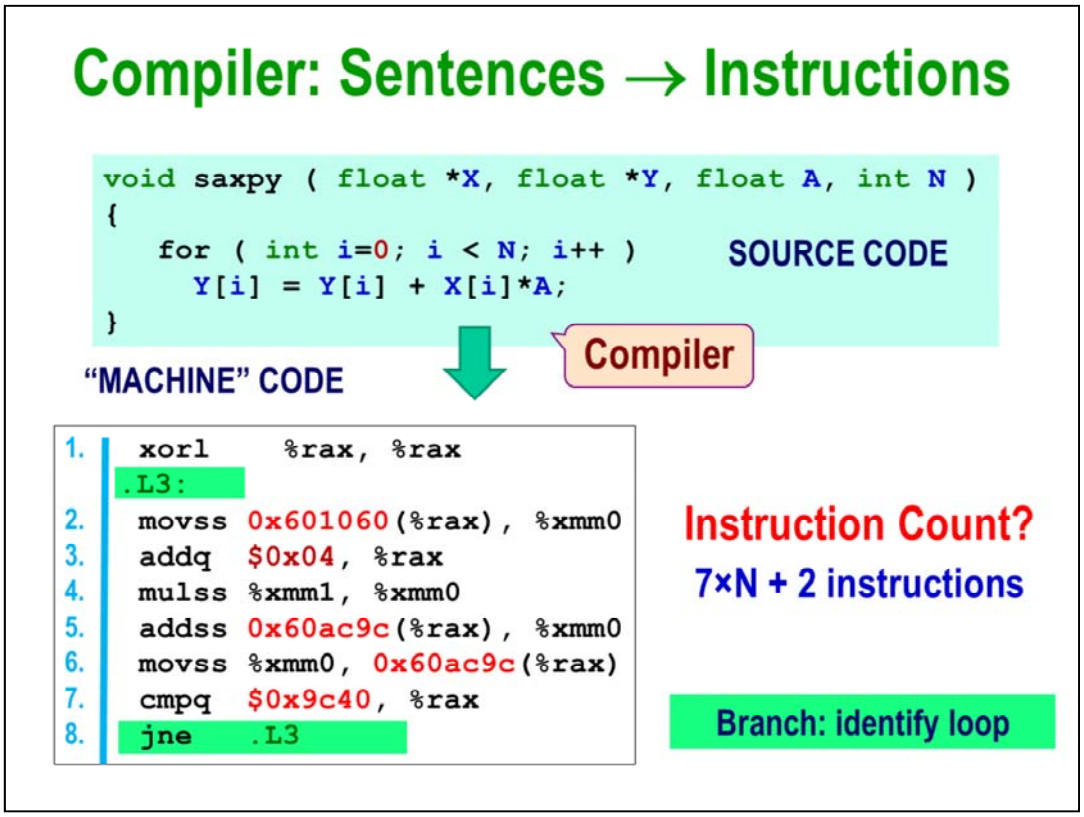
How many machine instructions per SAXPY op.?

Performance *profiling* & Assembly Code analysis

```
$ perf record
    ./saxpy
$ perf report
```

```
33,74 |18:   movss  0x601060(%rax),%xmm0
 0,03 |      add    $0x4,%rax
      |      mulss  %xmm1,%xmm0
 0,16 |      addss  0x60ac9c(%rax),%xmm0
32,35 |      movss  %xmm0,0x60ac9c(%rax)
33,66 |      cmp    $0x9c40,%rax
      |    ↑ jne    18
```

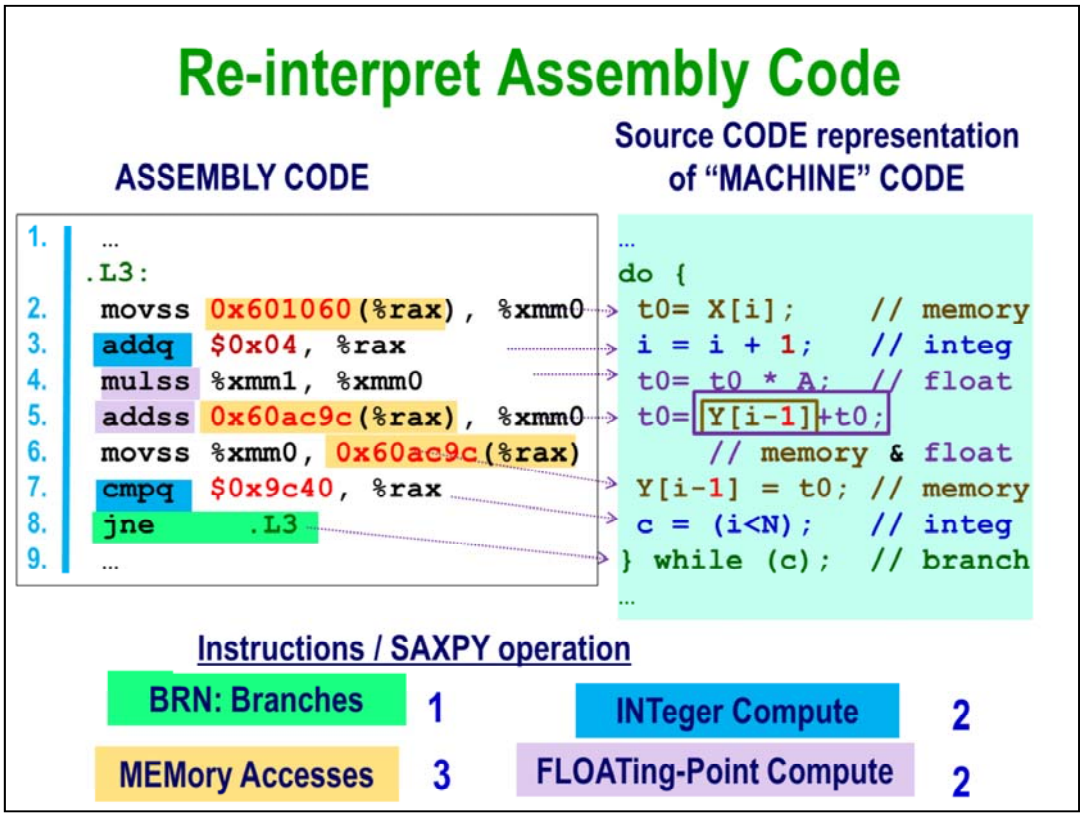
The slide shows how to invoke the profiler to visualize the machine instructions (in assembly language) responsible of most of the execution time taken by the program's execution (those instructions depicted in red are the "responsible" of 33.7%, 32.3% and 33.7% of the execution time). The assignment of time made by the profiler is very approximate, and it is only a hint of where execution time is being devoted. The assembly code represents the machine language: each assembly code line represents a single machine instruction. We will analyze this code in the next slide, which corresponds to the inner loop of the saxpy program.



Machine instructions are executed sequentially until some branch instruction makes the control flow diverge to a different point in the code. Most of the time, branches are used to go backwards in the code and implement loops, like in the example before. You can recognize branches because they often start with b (of branch) or j (of jump), and because they are followed by a label (like .L3) which also appears at the beginning of some empty line in the assembly code (line between 1 an 2).

The conditional branch at the bottom of the code, which branches or not depending on the outcome of some condition, determines a loop body composed of 7 instructions (from lines 2 to 8, including the branch), which we assume are executed N times (look at the high-level code above to understand what the program does). Therefore, we can deduce from the assembly code that executing this code fragment will involve the execution of 7N+2 instructions. Since we are doing engineering and not mathematics, we can approximate the number of instructions as 7N instructions.

This value confirms the program coding efficiency computed from the total number of machine instructions executed measured by perf

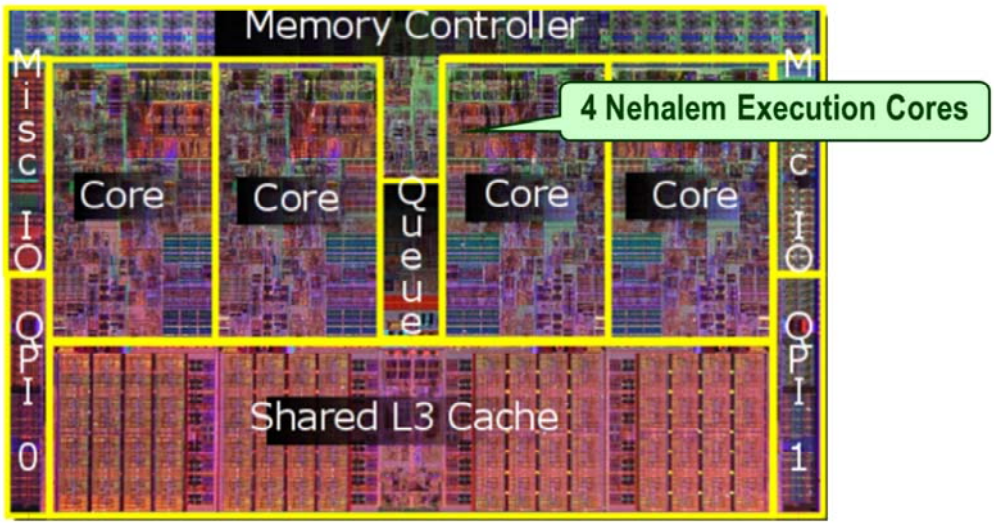


This slide shows a re-interpretation of the machine code in terms of high-level sentences in C language. The items starting with % in the machine code represent registers and can be defined in C language using local variables. The items enclosed by parenthesis in the machine code represent memory accesses, and can be described in C language as accesses to vectors (like X[i] o Y[i]). Remember that registers are very fast especial memory locations, and that there are so few registers that each has a name (like %rdi, %rax, $xmm1, %xmm0 …). The processor in the laboratory contains a total of 32 registers per computing core.

It is not the aim of this lesson to teach you how to fully understand assembly language, but for simple examples you can recognize multiply, addition and compare operations in the assembly code (mulss, addss, addq, cmpq) that can help you estimate how the compiler is converting the C sentences into machine instructions. In the example, it is not an extremely complex task to identify the 2 operations using single precision real (float) numbers (mulss and addss), the 3 memory accesses (read X[i] and Y[i] and write to Y[i]), the update of the loop index/counter (addq), and the comparison used to check for the loop termination (cmpq). Recall that you have the source code in C language that can help you understand the corresponding assembly code.

Therefore, 7 instructions per saxpy operation seems a fairly reasonable case. We will see soon that the use of vector (or SIMD) instructions can further reduce the total number of executed instructions.

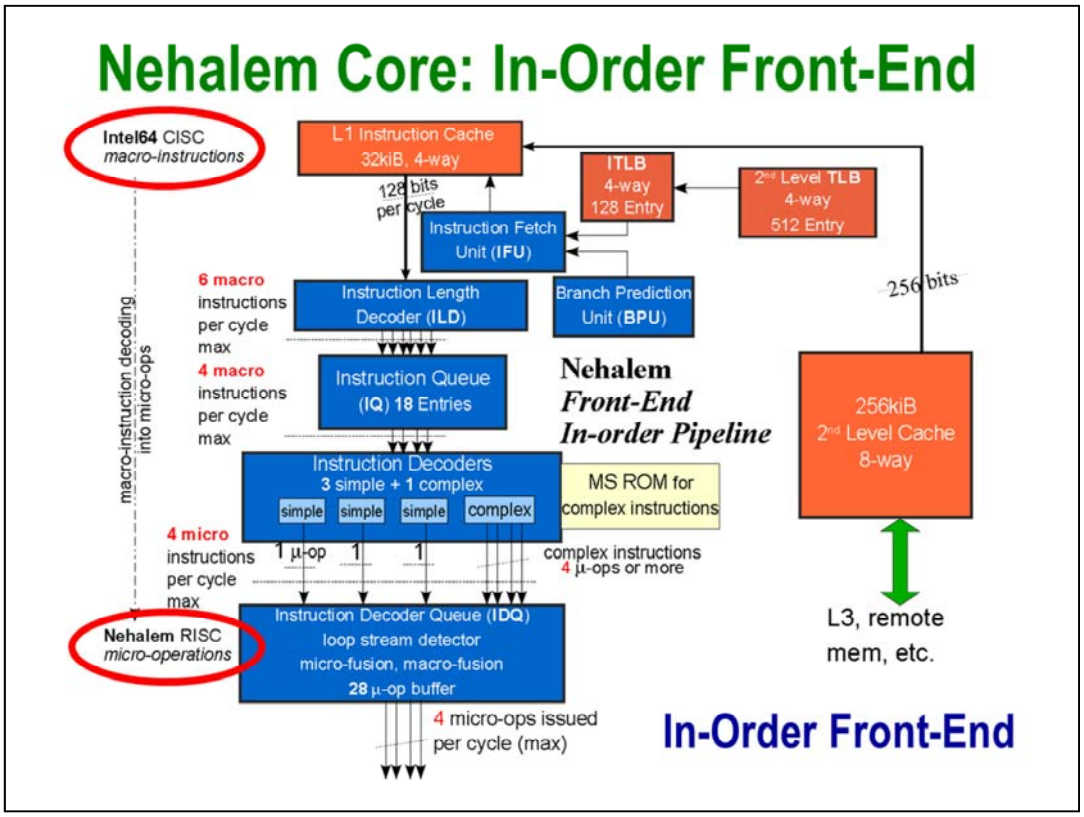
Processor: Intel i7 950 Nehalem (Lab)



http://ark.intel.com/es-es/products/37150/Intel-Core-i7-950-Processor-8M-Cache-3_06-GHz-4_80-GTs-Intel-QPI

The processor of the computers on the AC Laboratory (yellow label!) is an i7 950 of Nehalem architecture. In the slide you can check that it contains 4 execution cores, each of which can execute a program (or thread) independently (in fact, each kernel can execute two programs - or threads - at the same time, but we will explain this much later).

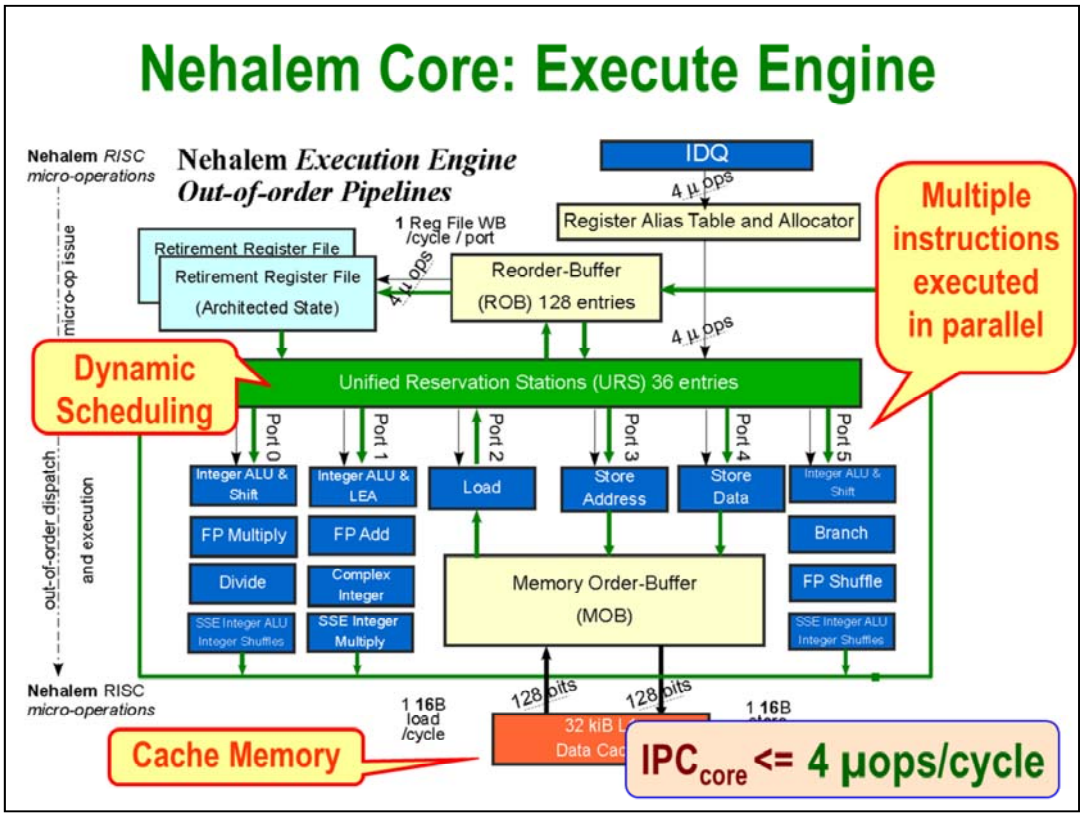
The most important thing to learn now is that: an execution core has all the components necessary to run a traditional sequential program. More than ten years ago, all processors were made up of a single execution core, and so there was no need to use the words “processing core”: a processor was a processor, period. Today, a processor can have 4, 8, 12 or more execution cores. It is very important to understand the difference between processor and processing core. From now on, and until further notice, we are always going to focus on a single core of computation.



The IPC measures the ratio of machine instructions executed per clock cycle. The higher the IPC, the better. A higher IPC indicates that the processing hardware is more efficient when executing the machine instructions. In order to explain the variations in the IPC it is necessary to understand the basics of the architecture of a modern processor: (1) processors are able to execute several machine instructions simultaneously, and (2) processors contain internal cache memory that is much faster than the external DRAM memory. We will start explaining the ability of processors to execute several instructions simultaneously.

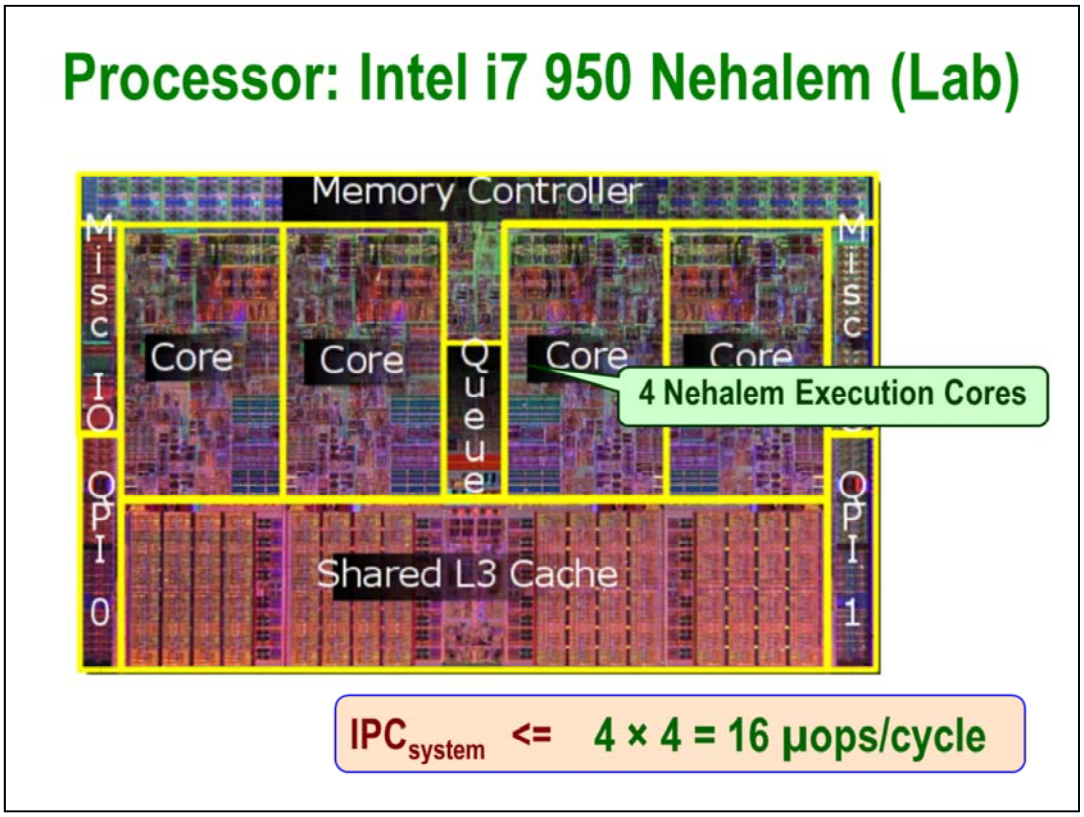
Each computational core of the Nehalem architecture consists of two basic blocks: the front-end and the execution engine. The slide shows the front-end, which processes the instructions in the same sequential order that the assembler program indicates. Instructions, each one codified using a variable number of bytes, are read from a special, small and fast memory, the instruction cache (above), and processed through the elements shown (top to bottom). In the end, what comes out of this front-end are µoperations, altogether a maximum of 4 each clock cycle. During this process the instructions are fetched from memory, decoded and converted to µoperations. The two elements that have the greatest influence on program performance are the instruction cache and the Branch Prediction Unit (BPU), which we will describe later.

What is important on this slide? To recognize the difference between instruction and µoperation, and to understand that there is a limit imposed by the H/W to the number of operations that can run each clock cycle, and therefore there is a limit to the IPC. In this particular processor the maximum IPC limit will always be less than or equal to 4: if each of the instructions in my program is converted to a single µoperation, then the IPC could become 4 (very difficult, but theoretically possible)- On the other hand, if, for example, every ten instructions are converted to 12 µoperations, the limit of 4 µoperations per clock cycle becomes an IPC limit of 10 * 4/12 = 3.333



We now present the "engine room" of a computational core of the Nehalem architecture: the so called execution engine. In this case, µoperations enter the engine and can be processed in a different order to the sequential order indicated by the assembler program. Execution can be reordered, so that certain operations may have to wait for multiple clock cycles in the Reservation Stations, while other operations are executed straightforwardly. The process of executing instructions out-of-order (or *Dynamic Scheduling*) will be studied later.

What is important on this slide, right now? Recognize the multiple execution units implemented in H/W and specialized for different types of operation. For example, there is a single H/W execution unit capable of processing µoperations of type LOAD. This implies that there is an additional limit of one LOAD operation executed per clock cycle. This limit can be critical for a certain program and determine the total execution time of the program. It is important to understand that the ability to execute each type of µoperation is an additional limit that we have to consider in our system. The operations that perform arithmetic-logical operations (ALU) of type INT can be processed in three different units (first, second and last). The operations that perform shift operations of type INT can be processed only in two different units (the first and the last), etc.

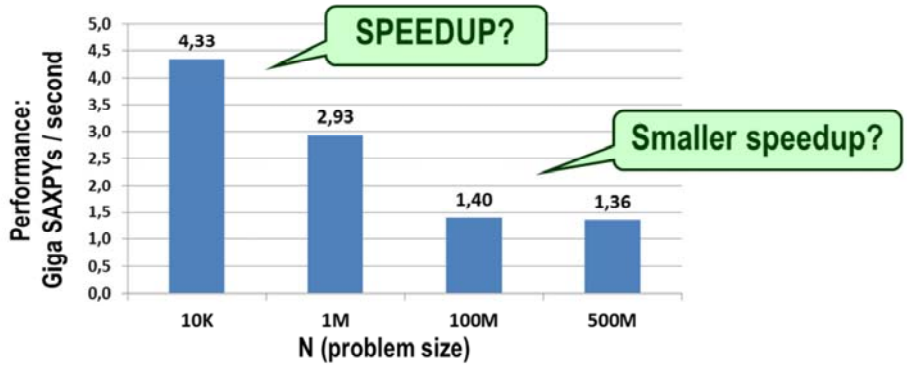


The whole processor is able to execute up to 4 instructions per cycle and processing core, which represents a potential of up to 16 instructions per cycle.

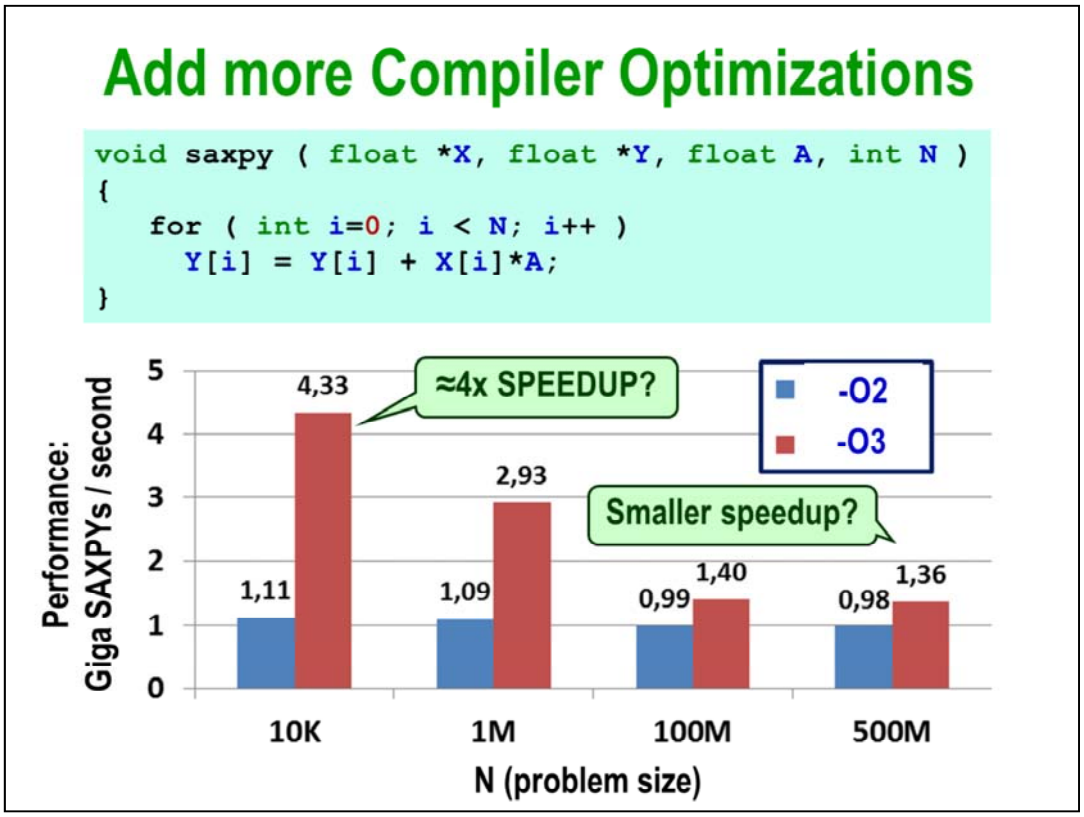
Add more Compiler Optimizations

```
void saxpy ( float *X, float *Y, float A, int N )
{
   for ( int i=0; i < N; i++ )
     Y[i] = Y[i] + X[i]*A;
}
```

```
$ gcc -O3 saxpy.c -o saxpy
$ perf stat -e instructions,cycles ./saxpy
```



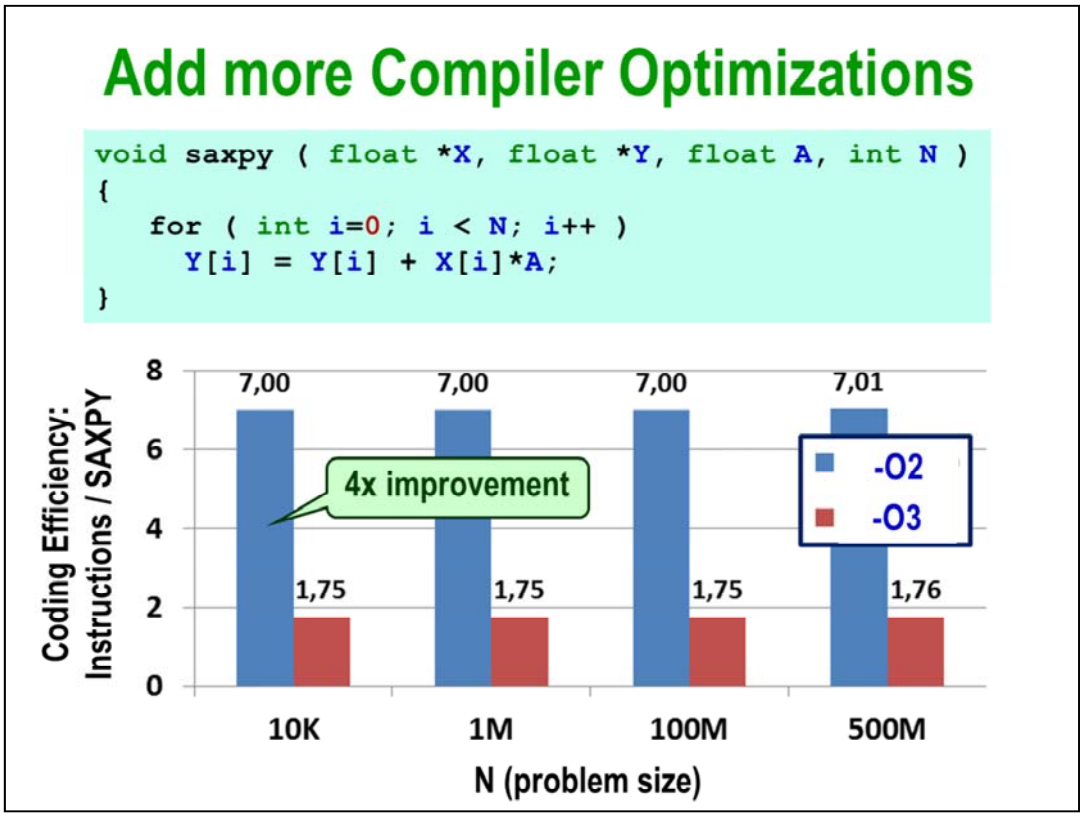
This slide shows the performance of the SAXPY program as the problem size grows, measured in (Giga) SAXPY operations executed per second. We can easily see that performance decreases as the problem size increases. But we cannot directly compare the improvement achieved thanks to using the –O3 optimization flag instead of the –O2 optimization flag. We should have included the performance results for both cases …



The slide now shows the performance of the SAXPY program as the problem size grows, measured in (Giga) SAXPY operations executed per second, for the two cases that we want to compare: using the –O2 optimization flag or the –O2 optimization flag.

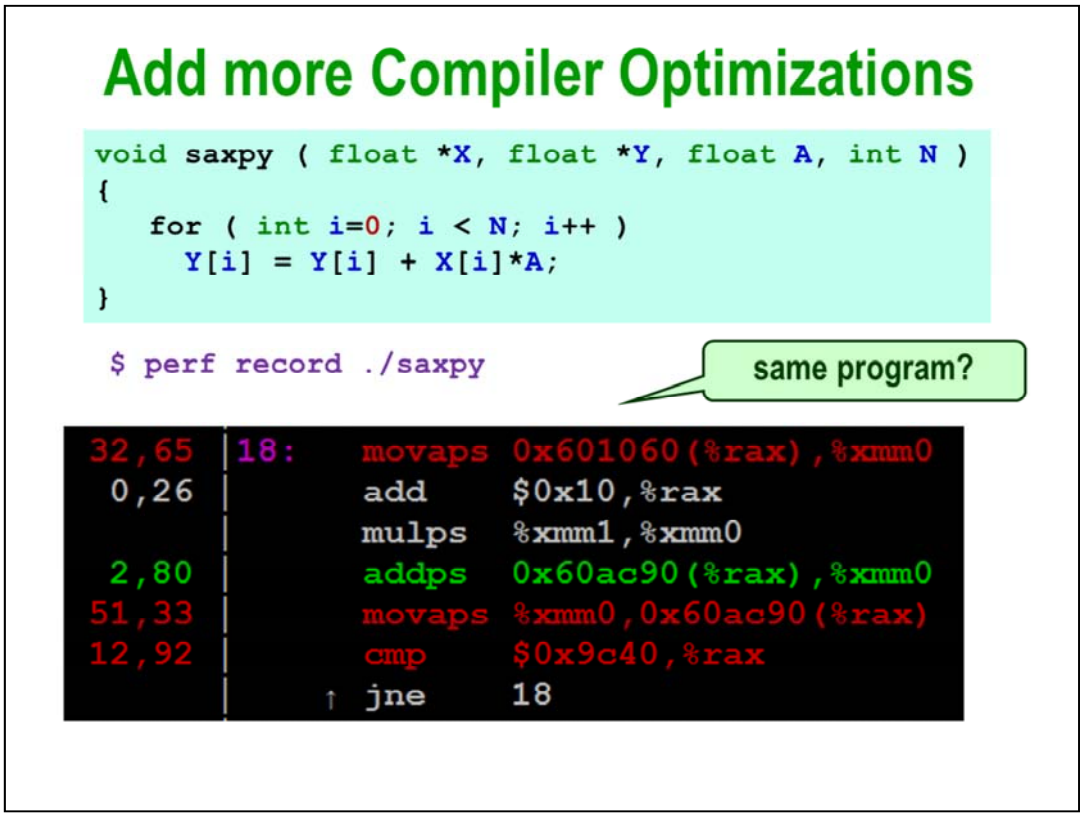
Now we can visually estimate the speedup obtained for a small problem size of N= 10K, which is around 4x. We need to explain what is the compiler doing in order to achieve such a high improvement, because sometimes the compiler will not be able to perform such an optimization without our help.

It is also important to notice how performance decreases when increasing the problem size more than 3x for the –O3 case, while the decrease in the –O2 case is much more slight (around 10%). We will explain this later in this document.



As explained in previous slides, performance is a function of processor raw speed (clock frequency), processor throughput (IPC) and program coding efficiency (high-level operations codified per machine instruction). Here we depict coding efficiency as the number of machine instructions per SAXPY operations (higher is better), to verify if the explanation of the performance improvement can be found here, and we can clearly see that the improvement in coding efficiency is 4x, exactly the same as the improvement on performance for a problem size of N=10K.

In other words, the program optimized by the compiler when the –O3 flag is used executes 4 times less machine instructions than the program generated when using the –O2 flag. What is the compiler doing? We cannot hardly can know the answer without looking at the assembly code.



This is the machine code generated for the saxpy C implementation when using the optimization flag –O3. At first sight, it seems almost identical to the previous code, but performance measurements reveal that the total number of machine instructions executed diminishes four times, from 7 instructions per SAXPY operation to 1.75 instructions per SAXPY. A superficial analysis of the machine (assembly) code does not explain the result. How can it be possible to load two values from memory, perform a multiplication and an addition and store the result with just 1.75 machine instructions?

The explanation requires understanding vector (or SIMD) instructions.

mulps: Multiply Packed Single-Precision Floating-Point Values

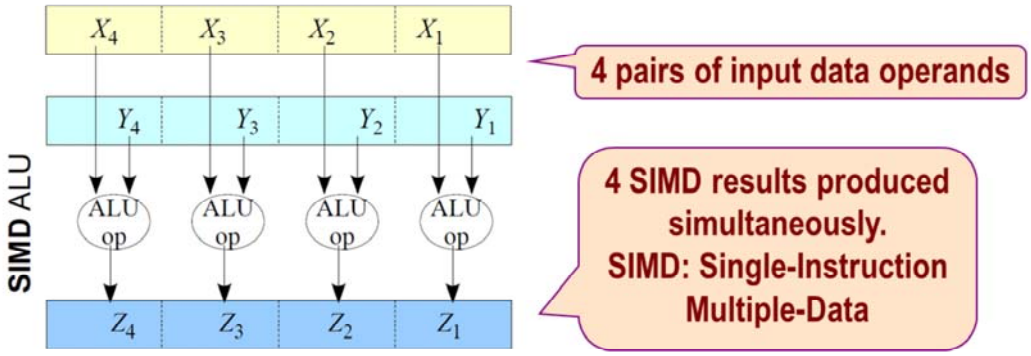
Opcode	Mnemonic	Description
0F 59 /r	MULPS xmm1, xmm2/m128	Multiply packed single-precision floating-point values in xmm2/mem by xmm1.

Description

Performs an SIMD multiply of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1 for an illustration of an SIMD single-precision floating-point operation.

Operation

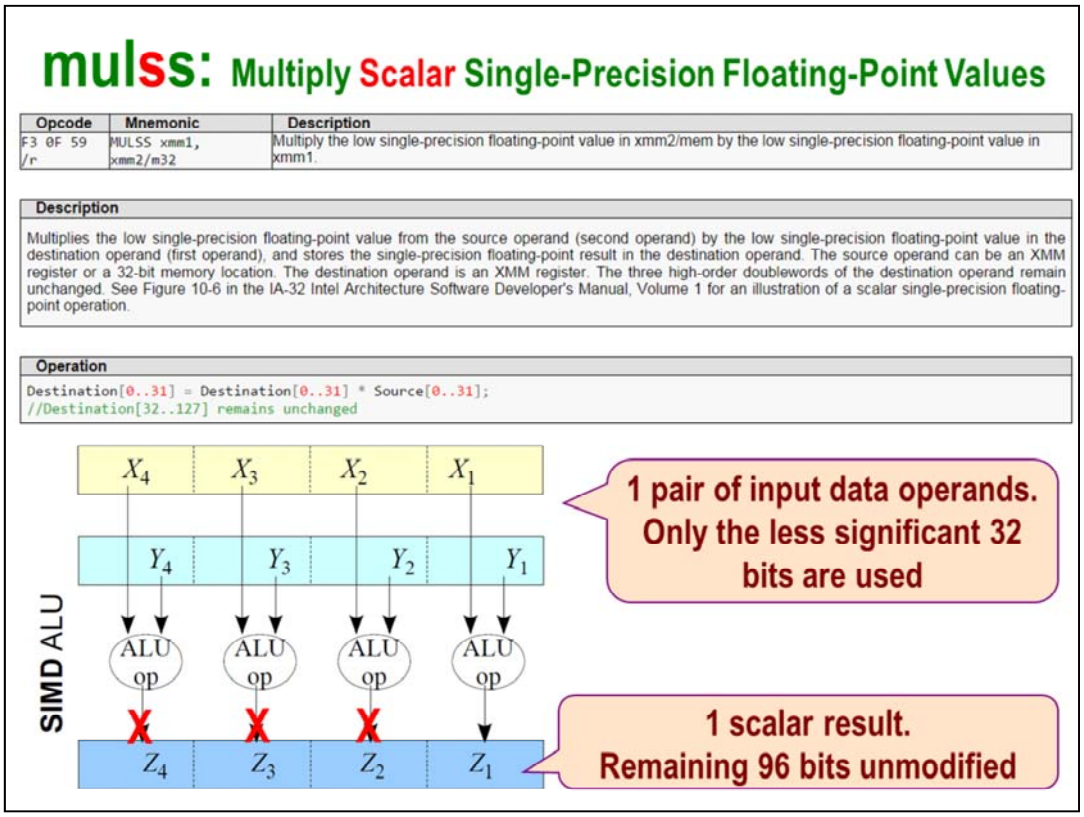
```
Destination[0..31] = Destination[0..31] * Source[0..31];
Destination[32..63] = Destination[32..63] * Source[32..63];
Destination[64..95] = Destination[64..95] * Source[64..95];
Destination[96..127] = Destination[96..127] * Source[96..127];
```



Searching the internet for the mulps instruction (http://x86.renejeschke.de/) we find the description above. If we understand the meaning of the description that is made in the last table, we can relate this explanation to the fact that the size of the registers %xmmN is 128 bits, even though we never do operations with such big numbers. In fact 128 bits are used to store 4 float data simultaneously (or 16 char data, or 4 int data, or 2 long int data, or 2 double data). Many documents refer to% xmmN registers as vector registers (because they store a small data vector) or SIMD (Single-Instruction Multiple-Data)

But in addition, the processor (or H/W) is able to do 4 operations simultaneously using the contents of two 128-bit registers to generate 4 results that will be stored in a 128-bit destination register.

What happens if I use the version of multiplication that has an s instead of a p?

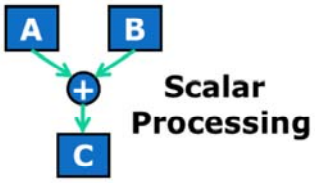


The mulss instruction has the above description, which indicates that only the least significant 32 bits of the input registers are used and only the least significant 32 bits of the target register are modified. Therefore, if the processor has the H/W capable of doing 4 operations simultaneously (SIMD = *Single-Instruction Multiple-Data*), and we only use one of the 4 "lanes" we have, we are using only 25% of the H/W potential ... and until now we did not know. And the processors of the future will offer more and more SIMD lanes, as we will see shortly.

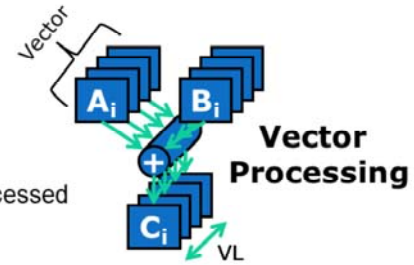
We can not happily give up the performance improvement offered by the SIMD instructions or vector instructions. Processors offer a limited subset of SIMD instructions: for example, it is not very useful to use instructions that operate with half the SIMD vectors. Thus, designers do not "spend" design time and transistors to implement instructions that would be used very little (it is always better to use the "complete" SIMD instructions and then calculate the operations that are left by using incomplete SIMD instructions in which only a single "lane" is used).

What is SIMD & Vectorization

- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD):
  - **Hardware supported** technique which allows an operation to be performed on multiple data points simultaneously.
  - Provides **D**ata **L**evel **P**arallelism (DLP) which is more efficient than scalar processing



- **Vector:**
  - Consists of more than one element
  - Elements are of same scalar data types (e.g. floats, integers, …)
- **Vector length (VL):**
  - Number of elements of the vector which are processed together

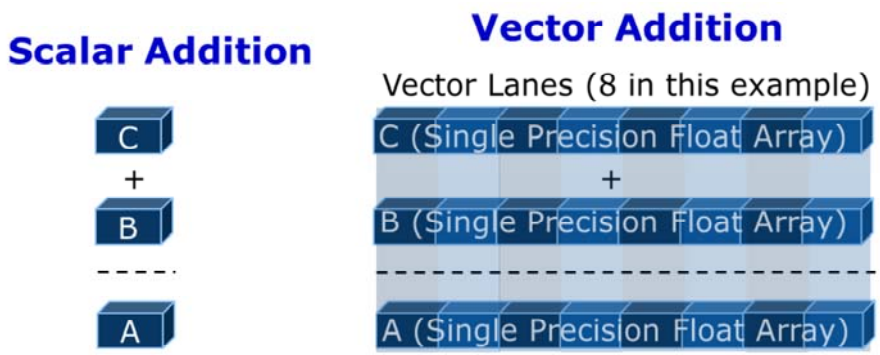


- **Vectorization**
  - Process which converts procedural loops that iterate over multiple pairs of data items and assigns a separate processing unit to each pair

Here we present some definitions, courtesy of an Intel presentation. The word vectorization is used to indicate the transformation of a scalar code (if you want, you can call it normal) in a code that uses SIMD instructions and that operates with small data vectors packaged in registers. Again, it is crucial to understand that the H/W of the current processors offers support for SIMD execution and, therefore, not using it means losing opportunities to increase free performance.

It is true that if the computation units and the SIMD registers are not used completely, a little energy consumption can be saved, but this consumption entails a very small amount of energy compared to what would be spent decoding and managing four times more instructions.

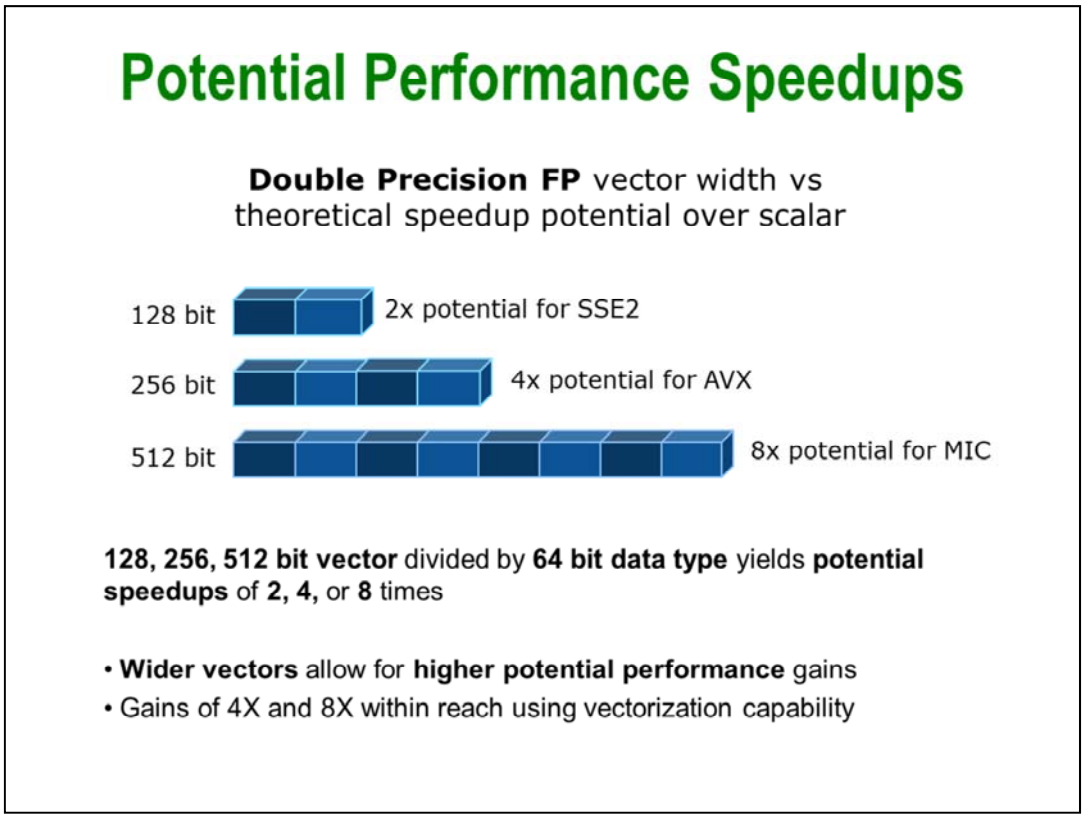
Example: Scalar versus Vector Addition



Vector length [in number of elements] =
size of vector register [in bits] / size of the data type [in bits]

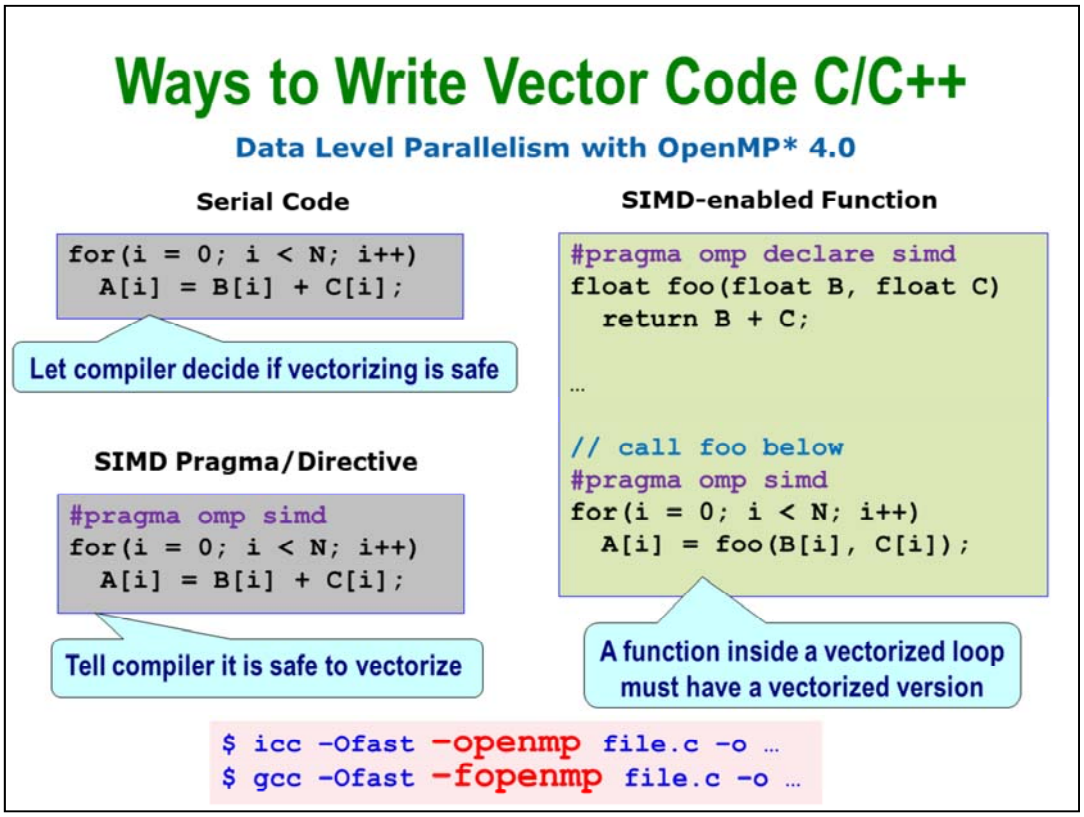
Number of **vector lanes** = Vector Length

The size of the %xmmN registers and the data we are using (char, int, double ...) determines the number of lanes we can use in an SIMD instruction.



The instruction set of the laboratory's processors implements SIMD instructions that are called SSE2, and correspond to the use of 128-bit registers. The most current processors have an extension to the x86-64 instruction set (ISA) that includes the use of 256-bit SIMD registers, which is called AVX. There are processors for high-performance systems, called accelerators and are an alternative to GPUs for these systems, which have the MIC instruction set, with 512-bit registers.

It should be noted that not all the operations that the processor is able to execute have their SIMD version, but only the most common (sum, multiply, or, and compare ...) or those that the processor designer considers to be more useful in the applications to which its processor is specially targeted.



You can use SIMD or vector instructions in your program by leaving the compiler to automatically optimize loops that you think are appropriate, or by helping the compiler with hints in the form of compilation directives. Compilation directives are the way to "talk" to the compiler: they start with #pragma and contain keywords specific to a certain compiler or standard, such as OpenMP (http://www.openmp.org/).

The first compiler directive used is **`#pragma omp simd`**, and tells the compiler that the loop that follows can be vectorized. Depending on the version of the compiler, this statement can be ignored, or the compiler may encounter problems in the data dependencies in the loop that prevent vectorization, or the compiler may not know how to generate efficient vectorized code. That is, the directive does not force vectorization but recommends vectorization.

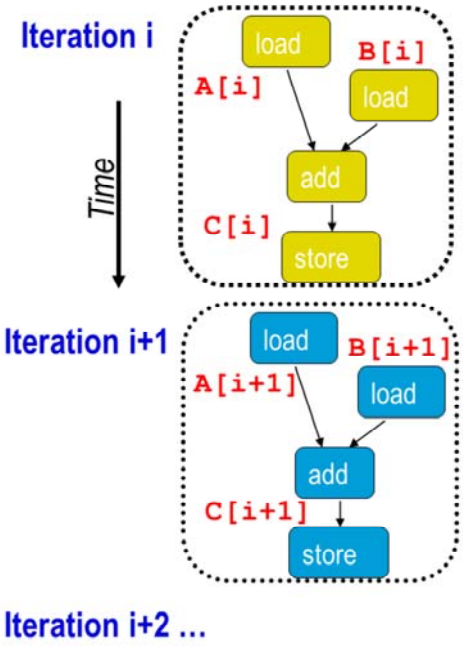
The directive **`#pragma omp declare simd`** tells the compiler to generate a SIMD version of the function that can be called from within a vectorized loop. In this example the vectorized function would receive as input two packaged data vectors in vector registers and would generate a packaged data vector in a %xmmN register.

Loop can be vectorized

Scalar Sequential Code

```
for (i=0; i < N; i++)
   C[i] = A[i] + B[i];
```

Vectorization requires *loop dependence* analysis



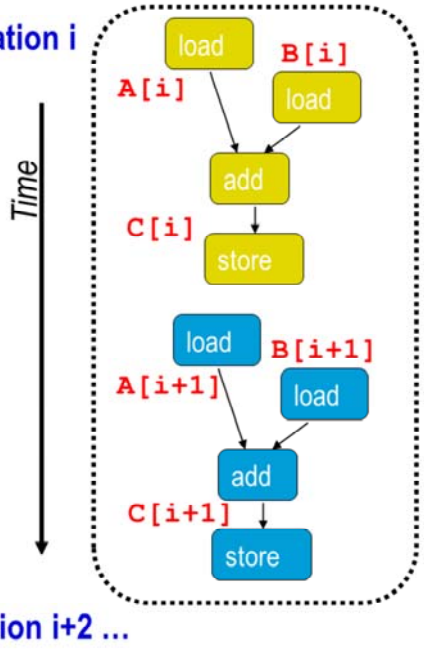
Therefore, using the **`#pragma omp simd`** directive does not ensure that the loop can be vectorized, and in many cases it is not necessary to use the directive for the compiler to be able to vectorize. In order to vectorize a loop it is necessary that the operations that are done in different iterations of the loop are independent of each other.

This slide shows the operations performed at each iteration of the loop, and you can see how the operations of the iteration i + 1 do not need the data generated by the iteration i. Therefore, the loop is potentially vectorizable.

First step: Loop Unroll

Unrolled Scalar Sequential Code

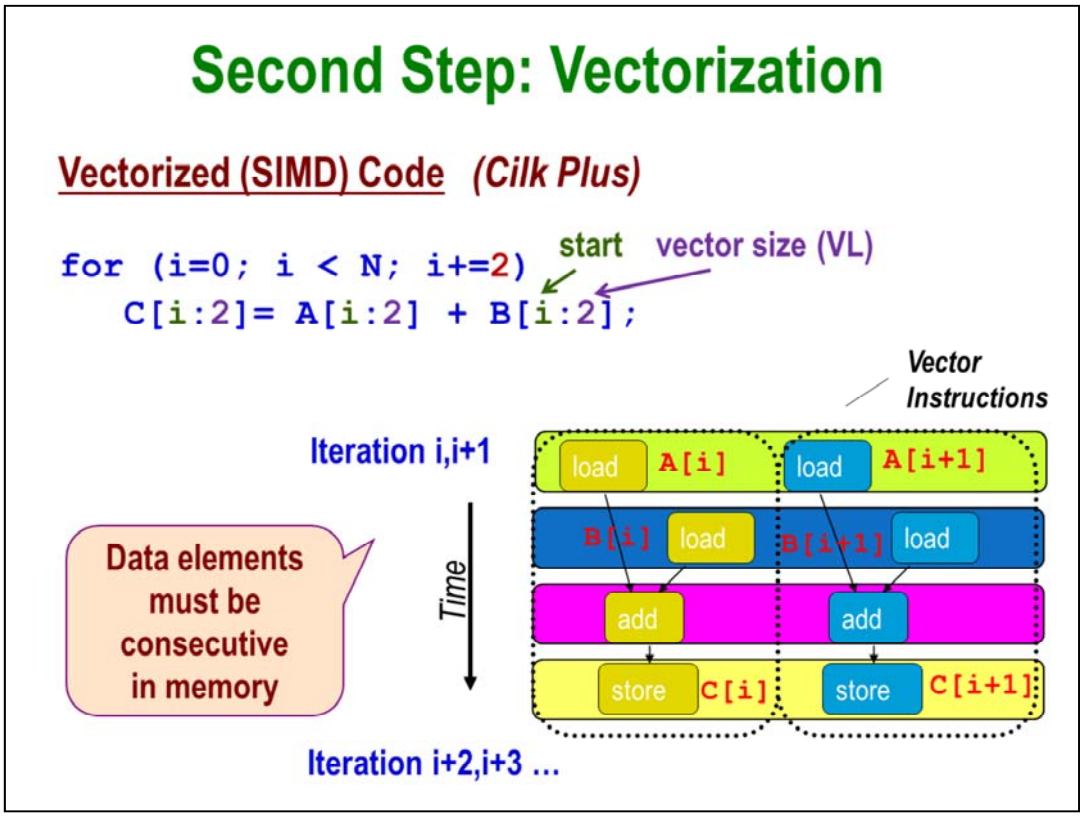
```
for (i=0; i < N; i+=2)
{
  C[i]   = A[i]    + B[i];
  C[i+1]= A[i+1] + B[i+1];
}
```



This slide shows the loop code which has been unrolled in two. Let's analyze the conditions necessary to be able to vectorize the loop efficiently:

- The two sentences inside the loop are independent: √
- Memory accesses are made to consecutive positions: √
- There is H/W support for the SIMD execution of the operations involved (FAD): √

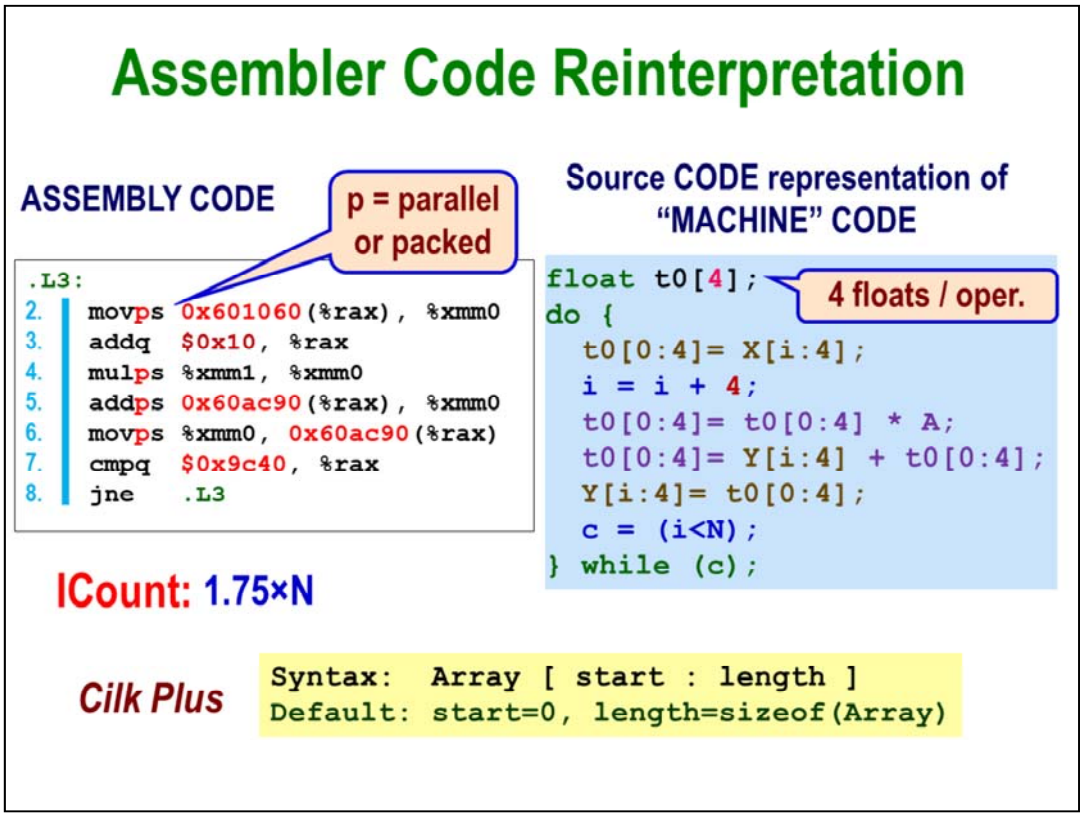
As all three criteria are met, these two statements can be merged into one using vector or SIMD operations, as we will show below.



There is an extension to the C / C ++ language, called Cilk Plus (cilkplus.org), which allows the explicit expression of vector or SIMD operations. To represent a sub-vector within a vector, the initial element and the number of vector elements are indicated in brackets.

In the slide, the vectorized code is expressed with Cilk Plus, assuming that each vector contains two elements. The drawing shows how the instructions for two iterations of the loop have been "collapsed", so that each instruction now doubles as before. We repeat: (1) in order to access two elements of memory with a single LOAD or STORE instruction , both elements must be consecutive in memory; and (2) in order to perform two calculations simultaneously with the two elements of the input vectors it is necessary for the processor to support that instruction in its instruction set.

At https://software.intel.com/sites/landingpage/IntrinsicsGuide you can find all updated H/W support information to SIMD instructions for the most current processors with the x86-64 set.



Now we can explain the resulting assembly code: the 'p' means parallel, and identifies the SIMD versions of the scalar instructions. Since every SIMD instruction handles a vector of 4 floating-point numbers, then instruction 3 adds number 0x10 (hexadecimal representation of the decimal number 16) to the register that indexes the memory locations (4 Bytes × 4 numbers = 16 Bytes) .

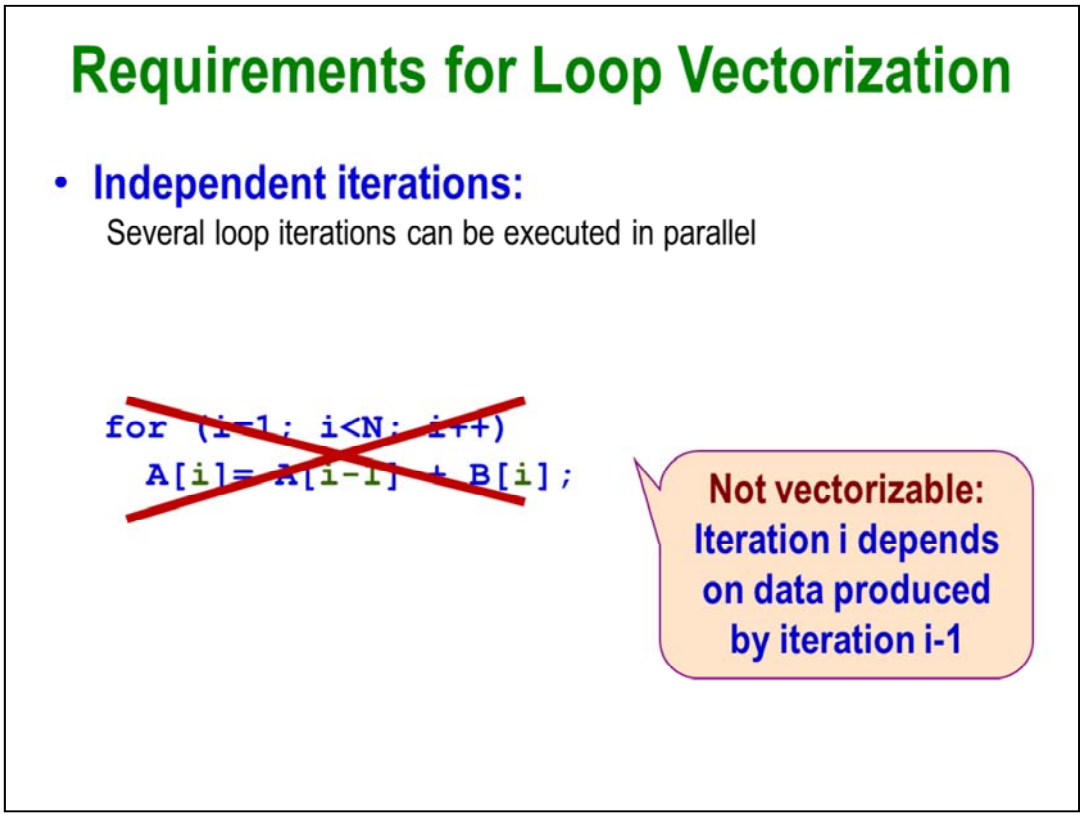
We can explain the performance of the code generated by the compiler with the -O3 option: 4 times less instructions are executed because the code is vectorized (uses SIMD instructions) and each load instruction (2 and 5), store (6), sum in floating point (5) and floating point multiplication (4) operates with 4 float data simultaneously. Since the vectorization is accompanied by a previous unrolling of the loop, which reduces the number of INT and BRN instructions, the final result is that the executed machine instructions are reduced 4 times for all types.

In many cases it is not possible to do a complete vectorization of a loop, and therefore improvements in performance are not always achieved in proportion to the number of lanes of the SIMD units.

The code equivalent to the assembler language is shown, where the first 4 SIMD instructions are represented using the Cilk Plus nomenclature. The colon symbol ":" is used to indicate that a range of vector elements is accessed. When the initial element is not indicated, it is assumed to be the first (index 0), and when the number of elements in the range is not indicated, it is assumed to be the number of elements from the initial element to the end of the vector.

Note that t0 represents a float data vector packaged in a 128-bit register. The syntax t0[:] indicates the access to the 4 elements of the vector at the same time, that is, access to the complete 128 bits.

The operation expressed in Cilk Plus as t0[:] = t0[:] * A, multiplies a vector by a scalar value, and involves multiplying each element of vector t0 by the value A, to obtain a result vector. The processor does not support operations between a scalar value and a vector, so the value A must be duplicated 4 times in the %xmm1 register before being able to do the multiplication. The initialization of this %xmm1 register must be done before entering the main loop.



The example code is not vectorizable, because each iteration of the loop depends on the previous one: at iteration i it is necessary to use the value A[i-1], which is generated just by the previous iteration of the loop.

If we tried to vectorize like this:

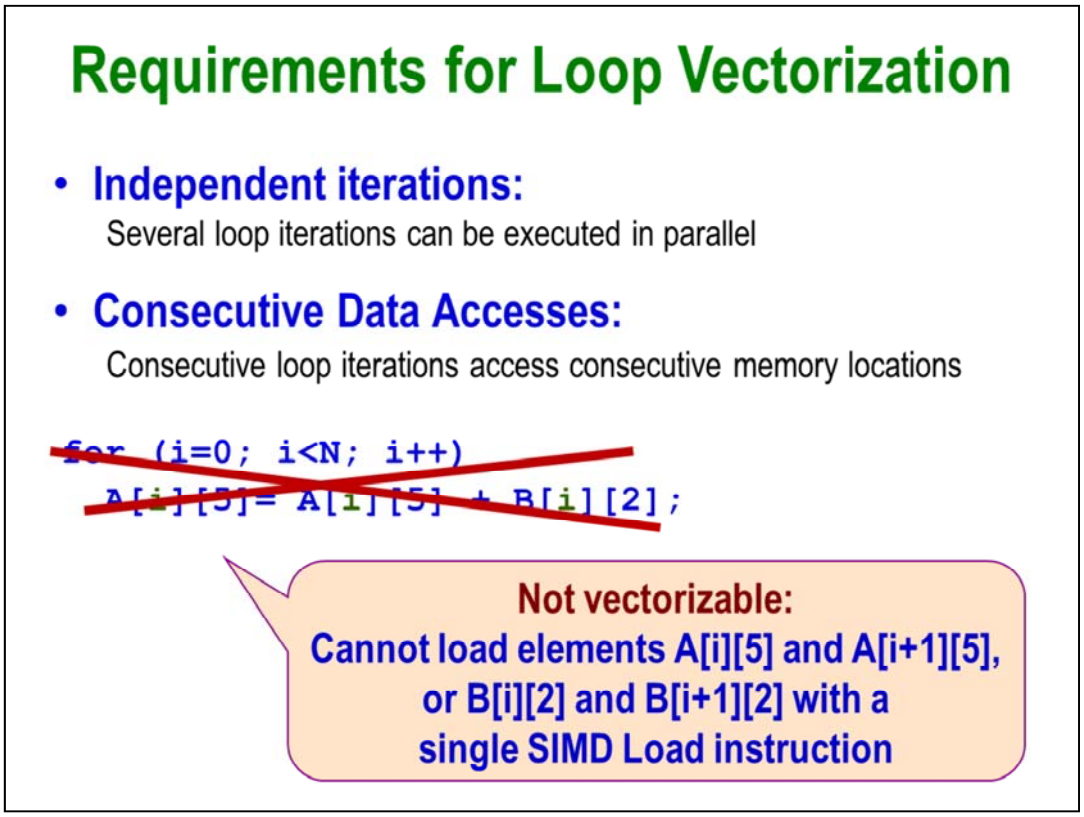
A [i:2] = A [i-1:2] + B [i:2]

We would not be producing the same results as in the original program. If you can not see that clearly, imagine that the values of the vectors are:

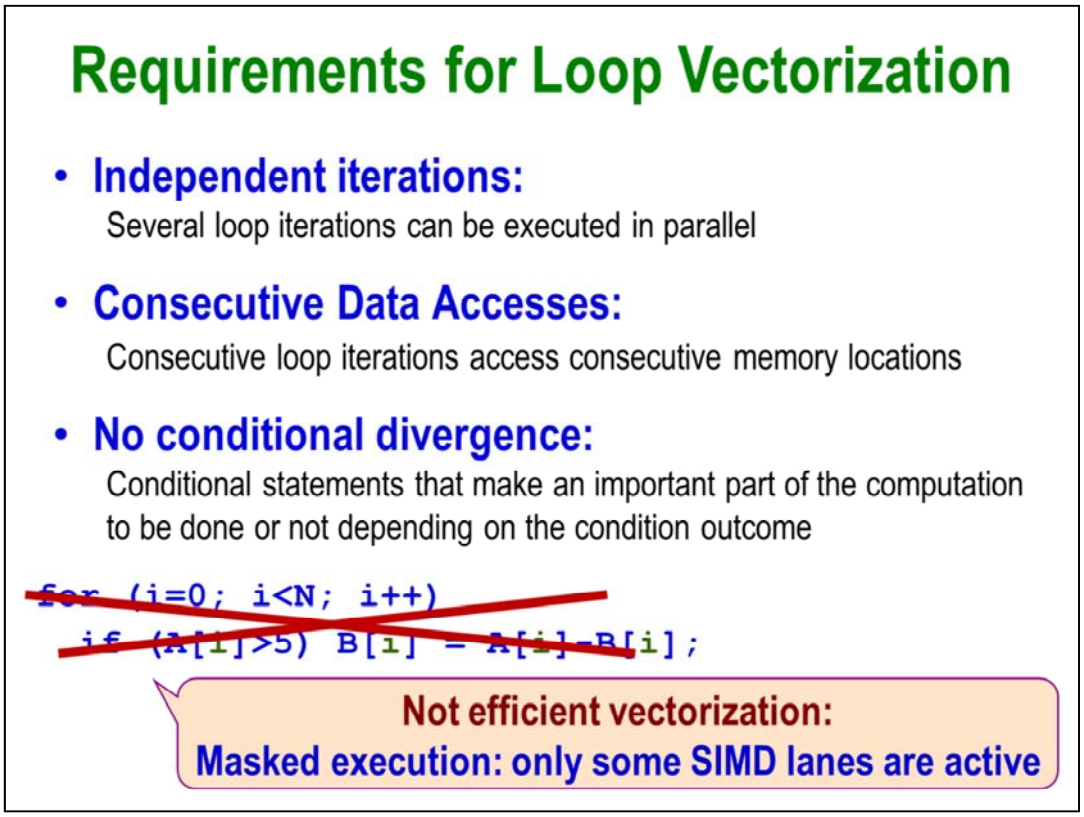
A = {1,2,3}; B = (3, 4, 5); N = 3

The original program would end with A = {1, 5, 10}

In contrast, the vectorized program as indicated above would end with A = {1, 5, 7}



The memory reads and memory writes of the sample code are not vectorizable because consecutive iterations access elements that are not consecutive in memory. This type of access is called *gather*, when they are read operations, and is called *scatter* when they are memory writes.



When the loop contains conditional statements, and the execution or not of certain operations depends on the input data, an efficient vectorization is not always possible.

A possible alternative is that in each iteration all operations are done in SIMD form, but finally, those cases where the condition is false are finally filtered, and in these cases the variables involved are not modified. That is, the most extreme case is assumed in which all the vectorized iterations have to do the maximum work, but then the updates of the variables are filtered to fulfill the conditions of the conditional sentence.

This alternative vectorization would only be efficient if the loop condition is true in a high percentage of cases.

Code Vectorization: Example 1

```
double VectorSum (double *In, const int N) {
  int i;  double sum = 0.0;
  for (i=0; i<N; i++)
     sum + = In[i];
  return sum;
}
```

Is this code vectorizable?

```
$ icc -Ofast -no-vec -unroll0  program.cc …
```

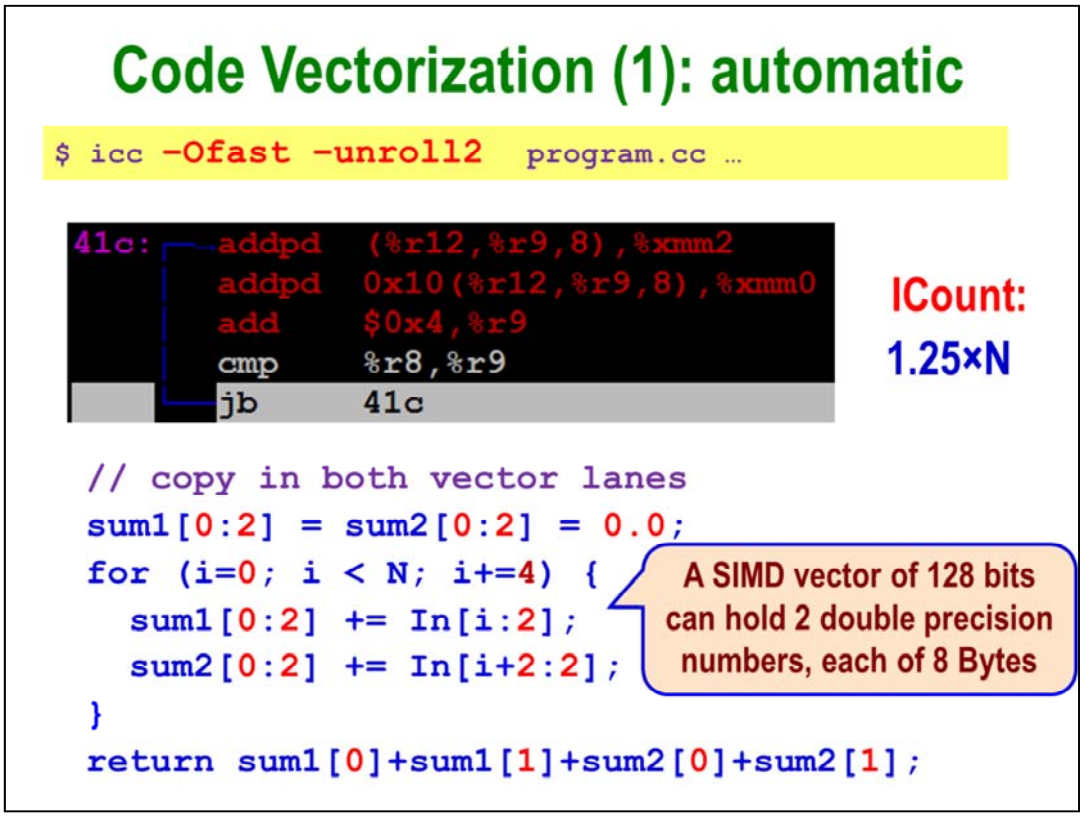
```
11c:  addsd   (%r15,%rdx,8),%xmm0
      inc     %rdx
      cmp     %r13,%rdx
      jl      11c
```

ICount:
4×N

The example code does not appear to be vectorizable, because each iteration of the loop depends on the previous one: in iteration i it is necessary to use the value sum that is generated just by the previous iteration of the loop.

The result obtained by explicitly indicating that the compiler should not vectorize nor unroll the loop (options -no-vec and -unroll0) is quite predictable: an instruction that does a LOAD and a floating-point sum (FAD), and the three instructions that control the loop. In total, 4 instructions per iteration.

But the floating-point addition operation is associative, so we can change the order in which we do the sums ...



If we now let the compiler vectorize and unroll the loop 2 times, we get a vectorized code: two SIMD statements that LOAD a pair of double elements and one SIMD floating point (FAD) sum. The combined result of vectorizing and further unrolling the loop is an improvement in coding efficiency (a reduction in the number of executed instructions) from 4 to 1.25 (3.33x).

The code below shows an equivalent way to represent the assembler program generated by the compiler.

How is it possible to vectorize, if the original program had a clear dependence on data? Because the sum operation is (theoretically) associative, and therefore we can change the order in which sum operations are made without changing (significantly) the result. Actually, you can slightly change the result of the sums represented in floating point (due to small rounding errors), and that is why we have to explicitly give permission to the compiler using the -Ofast flag.

Further note that the compiler uses different SIMD registers for each of the two machine instructions (%xmm0 and %xmm2), which makes both instructions independent and potentially can be executed in parallel.

Code Vectorization (1): Directives

```
double VectorSum (double *In, const int N) {
  int i;  double sum = 0.0;
  for (i=0; i<N; i++)
     sum + = In[i];
  return sum;
}

double VectorSum (double *In, const int N) {
  int i; double sum=0.0;
  #pragma omp simd reduction(+:sum)
  for (i=0; i < N; i++)
    sum += In[i];
  return sum;
}
```

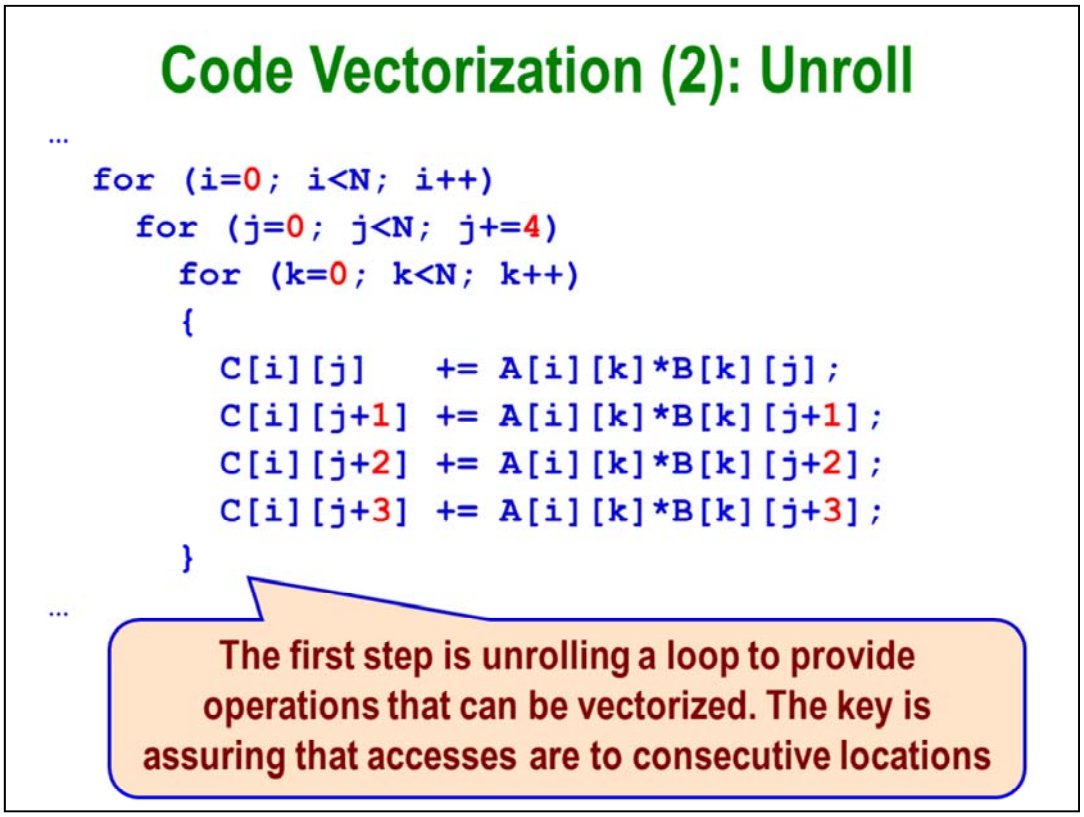
Tell compiler to vectorize reduction pattern

This slide shows a compilation directive that tells the compiler that the loop is vectorizable by means of a reduction type operation, involving the sum operator and the sum variable. This code example does not require this directive, because the code is very simple and the compiler can analyze it automatically, but in other cases it can help the compiler to improve its work.

Code Vectorization: Example 2

```
void MatrixMult ( float *A, float *B,
                  float *C, const int N) {
  int i,j,k;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        C[i][j] += A[i][k]*B[k][j];
  return;
}
```

Can you show how to vectorize the code using the Cilk Plus vector syntax? Assume SIMD vectors of 128 bits



If the intermediate loop is unrolled four times, as seen in the code, the accesses that are made in the four statements that have been unrolled (inside the inner loop) are to consecutive positions (in matrices B and C) or to the same position (in matrix A). This will allow you to directly use SIMD-type, LOAD and STORE instructions to access the data.

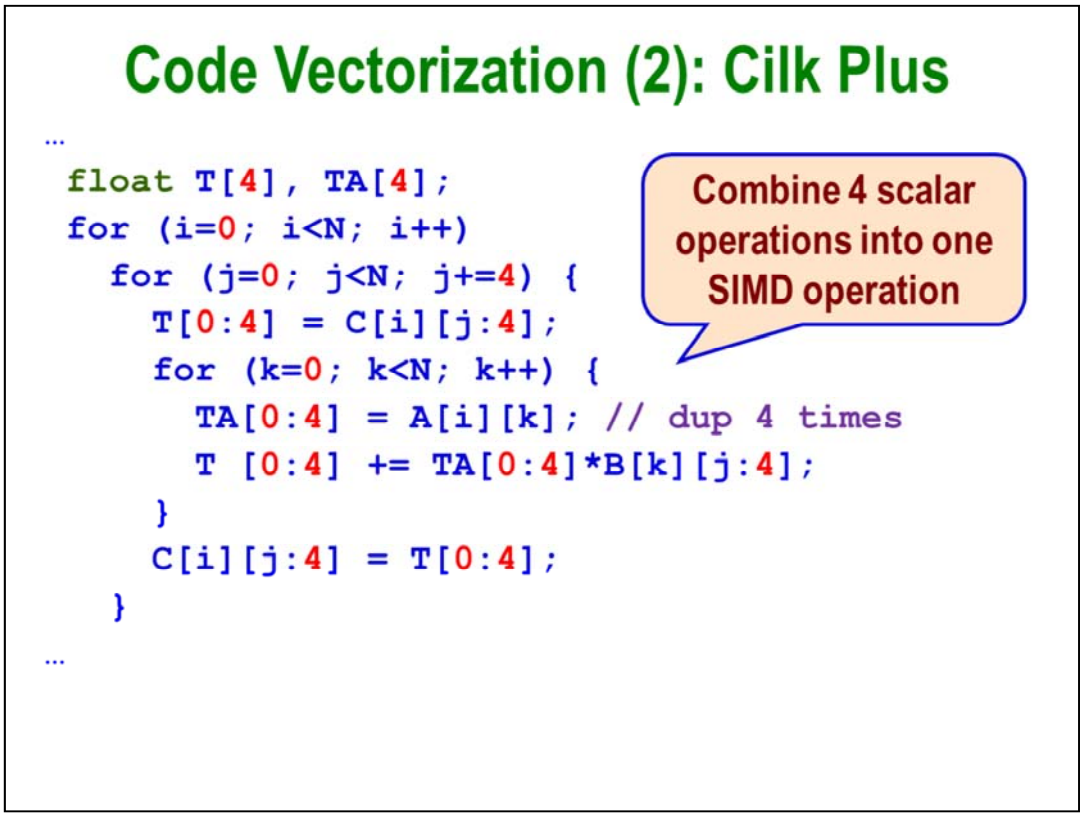
Code Vectorization (2): local variables

```
...
 for (i=0; i<N; i++)
   for (j=0; j<N; j+=4) {
     T0= C[i][j];   T1= C[i][j+1];
     T2= C[i][j+2]; T3= C[i][j+3];
     for (k=0; k<N; k++) {
       TA = A[i][k];
       T0 += TA*B[k][j];   T1 += TA*B[k][j+1];
       T2 += TA*B[k][j+2]; T3 += TA*B[k][j+3];
     }
     C[i][j]=   T0; C[i][j+1]= T1;
     C[i][j+2]= T2; C[i][j+3]= T3;
   }
...
```

Use local variables to reduce memory loads and stores

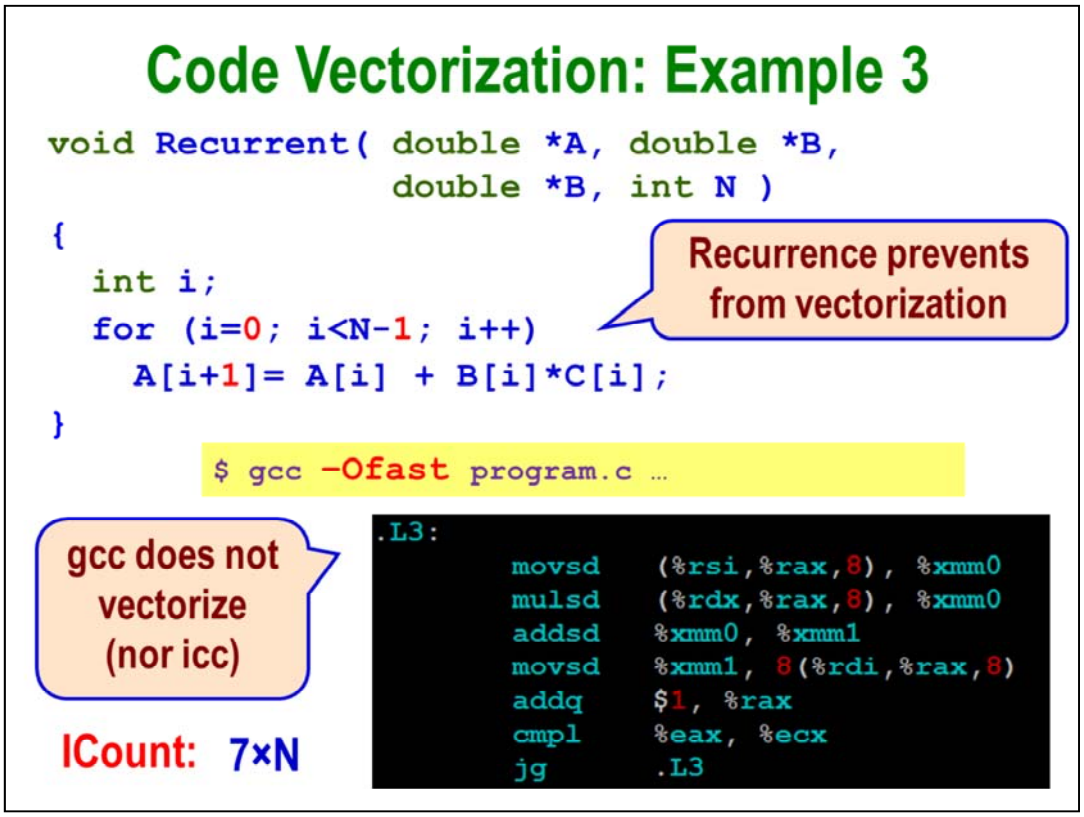
Before vectorizing, we show an optimization that consists of using temporary variables to store values that are going to be reused, and thus avoiding to access memory more times than necessary. For example, the variable TA stores the scalar value read from matrix A and is used four times within the internal loop. The variables T0, T1, T2 and T3 contain the values corresponding to the elements of matrix C that are being updated. With these 4 variables 4 reads and four writes to memory are avoided in each iteration of the internal loop.

The result is that the internal loop reads 5 different data in each iteration, which is equivalent to 5/4 = 1.25 LOADs per addition-multiplication operation of the original algorithm, whereas in the initial algorithm 3 LOADs and 1 STORE were done by addition-multiplication operation.

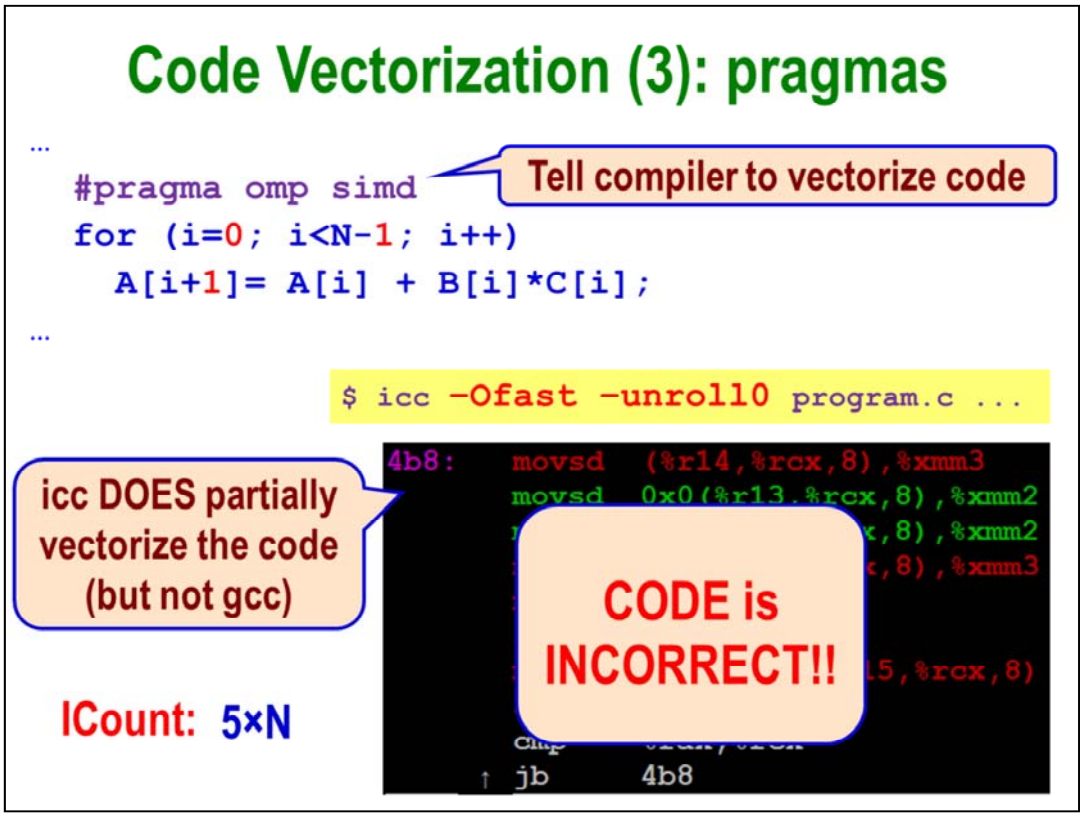


Finally, the operations are grouped four by four: variables T0, T1, T2 and T3 are put together in a single vector T[4] of 4 elements. The TA variable is duplicated in 4 identical copies of the same value.

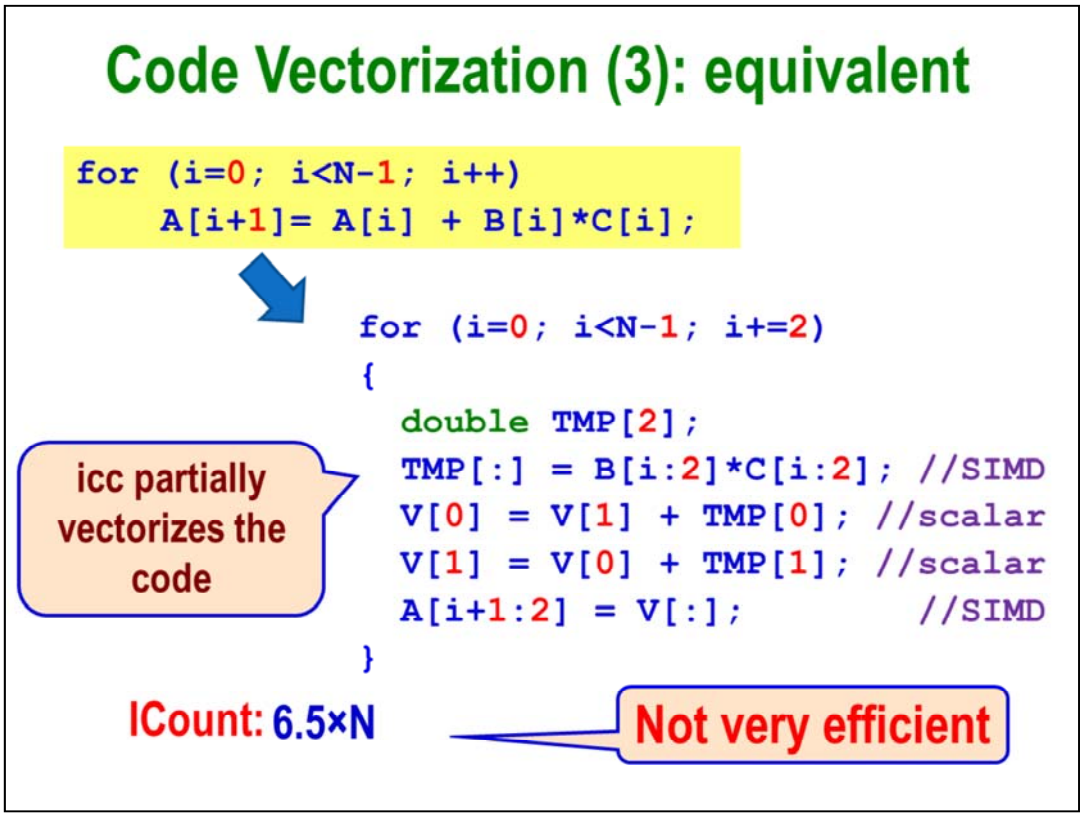
The result is not only a vectorized code, but also a code that has reduced the total number of memory accesses.



Neither compiler is able to automatically vectorize the code, because the loop iterations are dependent.

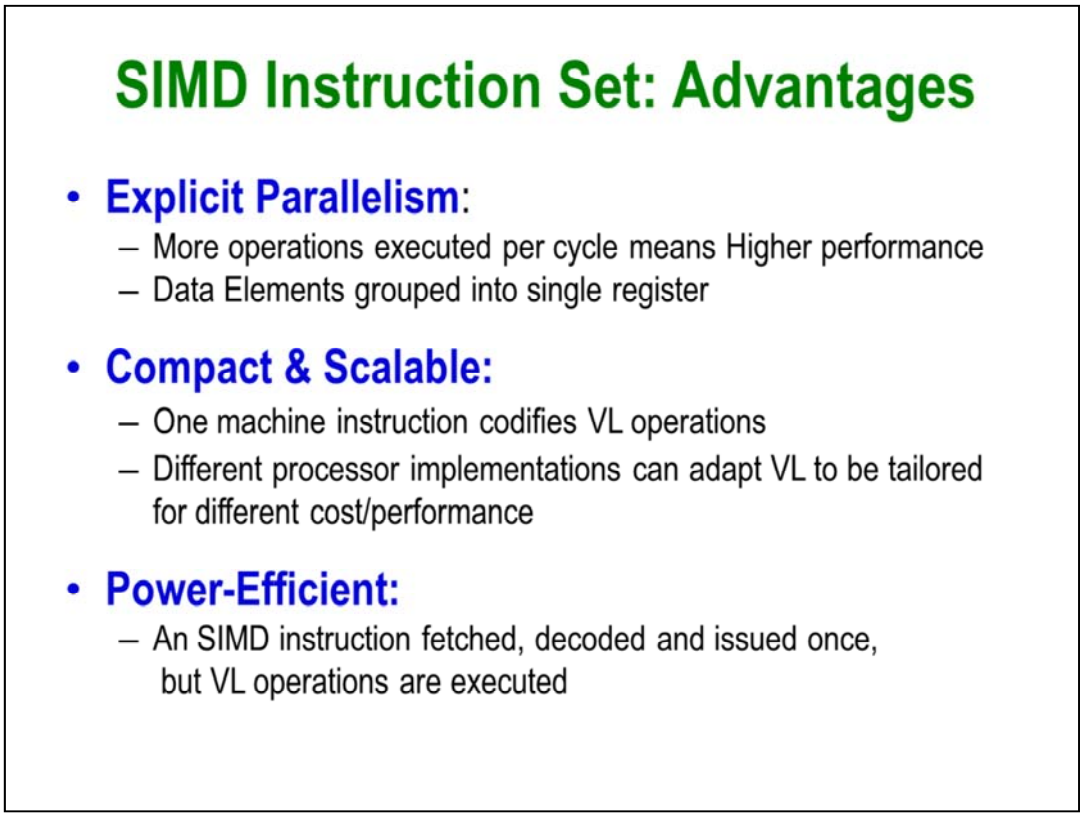


If we force the compiler to vectorize the loop, what we get is better performance ... but with an incorrect program. Therefore, we must be careful with the use of the directive. In addition, whenever we modify the code we must make sure that the result it produces is still correct.



If we modify the code by hand to show the compiler which parts of the loop can be vectorized, the compiler generates a correct program, but is not able to generate a very efficient code. Currently, compilers are not able to generate a very efficient code from the specification that uses Cilk Plus.

Summary: there are algorithms where vectorization is complex, and the compiler does not know how to generate the best possible code; in these cases the programmer can make a difference.



SIMD (Single-Instruction Multiple-Data) parallelism is a very inexpensive way in energy consumption and in silicon area consumption to increase program performance. The big problem with this kind of parallelism is ... the programmability.

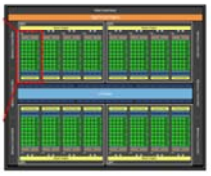
One could expect that the use of SIMD parallelism would be managed directly by the compiler, and would free us from having to use it explicitly. But using SIMD parallelism can be as difficult or more than using the parallelism of threads to take advantage of the multiple computation cores of a processor. By the way, the parallelism of threads is called MIMD (Multiple-Instruction Multiple-Data).

SIMD Applications

- **Scientific High Performance Computing (HPC)**
- **Multimedia Processing** (compress., graphics, audio synth, image proc.)
- **Standard benchmark kernels** (Matrix Multiply, FFT, Convolution, Sort)
- **Lossy Compression** (JPEG, MPEG video and audio)
- **Lossless Compression** (Zero removal, RLE, Differencing, LZW)
- **Cryptography** (RSA, DES/IDEA, SHA/MD5)
- **Speech and handwriting recognition**
- **Operating systems/Networking** (memcpy, memset, parity, checksum)
- **Databases** (hash/join, data mining, image/video serving)
- **Language run-time support** (stdlib, garbage collection)

To emphasize the importance of the use of SIMD instructions, here is a non-exhaustive list of applications that can be vectorized and therefore improve their performance considerably and with a minimum increase in energy consumption.

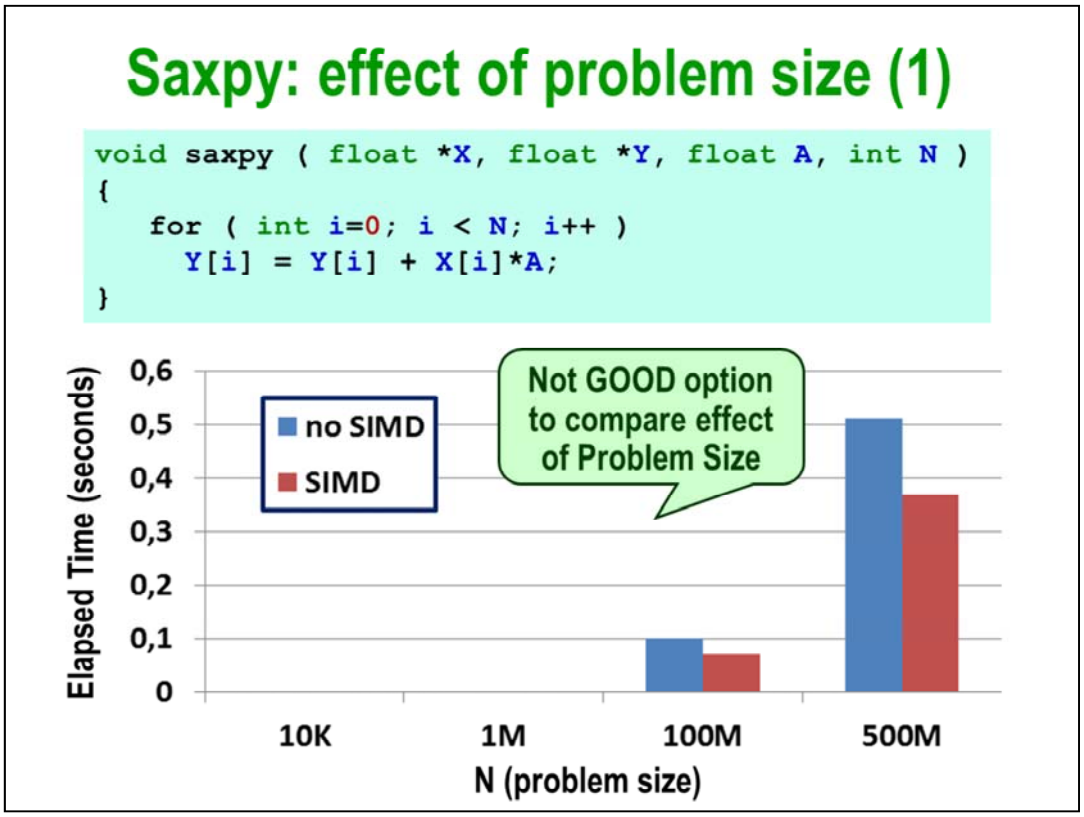
Processor Architecture



Cache Memory Hierarchy

As we have explained before, a performance analysis requires relating results with the way a computer works. Memory performance depends on the hierarchical organization of the different memory storage elements of the processor.

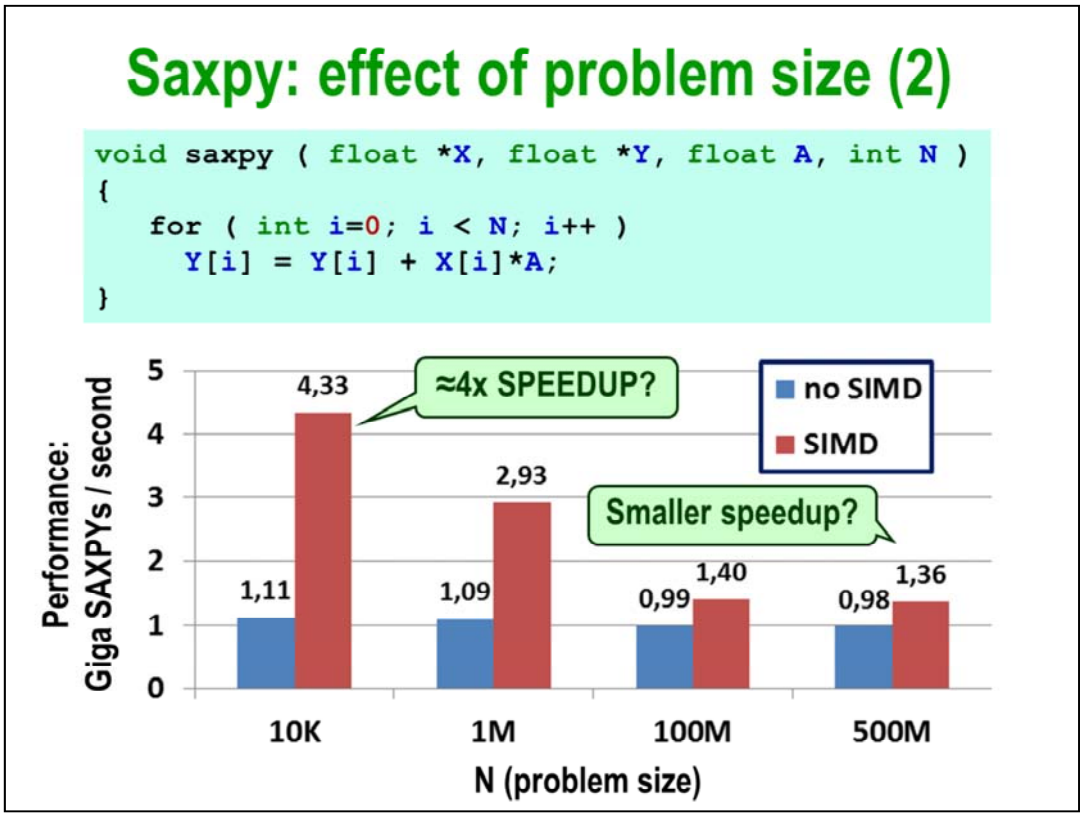
Again, the expectation is that a good diagnostic of the reasons of the performance results will straightforwardly lead us to a battery of code optimizations that will allow us to achieve better performance outcomes.



We show the elapsed time of the SAXPY program for different problem sizes (in this case the size of the problem is defined by N, which is the size of the vectors), and for two cases: non-vectorized code and vectorized code.

Although the chart is formally correct, it does not meet its reporting purpose adequately. First, the values for N = 10K and 1M do not appear in the graph because they are too small. Second, it is difficult to compare the results of N = 100M and N = 500M: the results of the blue bar (not SIMD) coincide well with the horizontal lines, and allow to calculate relatively easily that the execution time with N = 500M is about 5 times greater than with N = 100M, which is expected, since the temporal complexity of the algorithm is linear with N. But it is not so easy to do this calculation with the values of the red bar.

The only interesting information that is clearly observed in the graph is that the SIMD version (red bar) is between 30% and 40% faster than the non-SIMD version (blue bar), for values of N = 100M and 500M. This improvement is not what we should expect, and we will analyze it below.



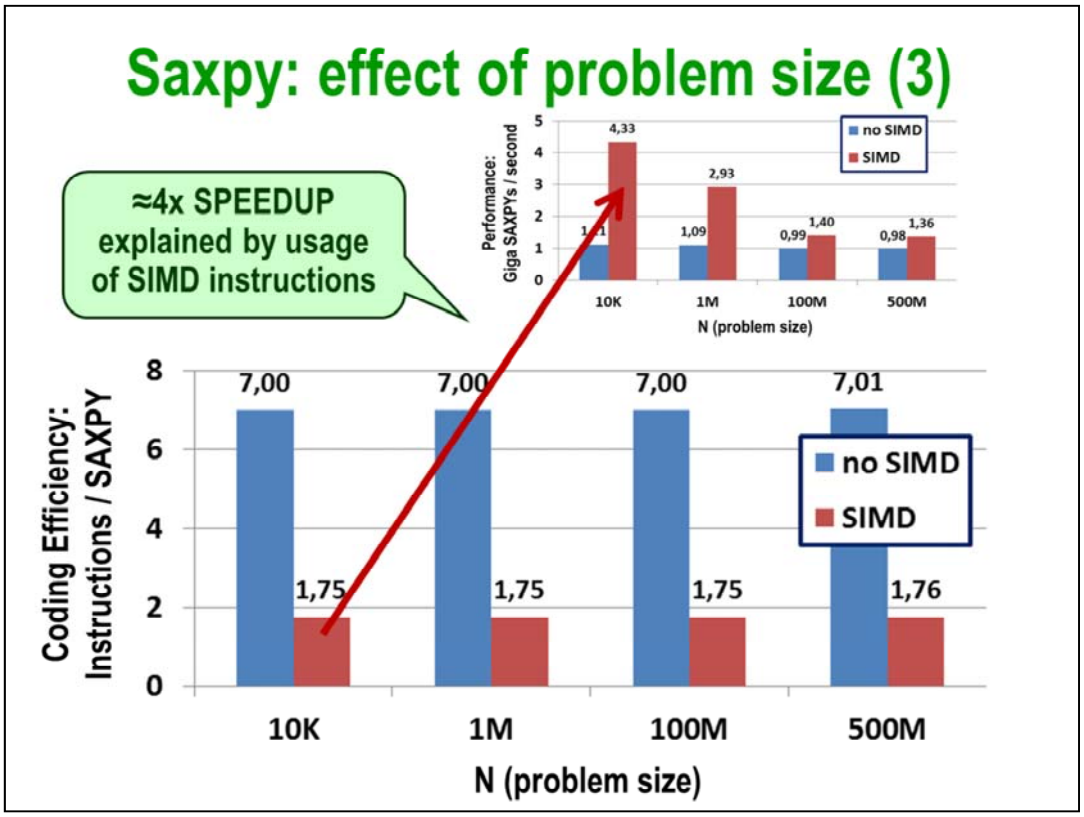
The graph we present now is much more significant: the performance expressed as work done per second, measuring the work in Giga SAXPY operations (billions). When using the units in Gigas we save to have to indicate 9 zeros in the units of the vertical axis (it is important to save unnecessary "ink"). On the other hand, normalizing the performance by the work performed avoids cases of results so small that are not visualized in the graph, and also facilitates the comparison of results for different problem sizes.

The first important result is that with N= 10K the improvement produced when using SIMD instructions is almost 4 times. This coincides with the value that we should expect, since the SIMD instructions process data 4-by-4 instead of 1-by-1.

The second, much more important, and motivating result for this subject is that performance slows down as the size of the problem grows. And the descent is much more pronounced for the code that uses SIMD instructions than for the one that does not use them.

As indicated above, the performance measured in operations per second can be "explained" as the multiplication of three factors, which also meet the “the higher the best" rule: (1) program coding efficiency measured as number of operations coded by machine instruction; (2) processor productivity or IPC (average number of instructions executed per clock cycle); and (3) the clock frequency of the processor (measured in giga cycles per second or GHz).

As the clock frequency of the processor is constant (or should be), we have two factors to explain the results.

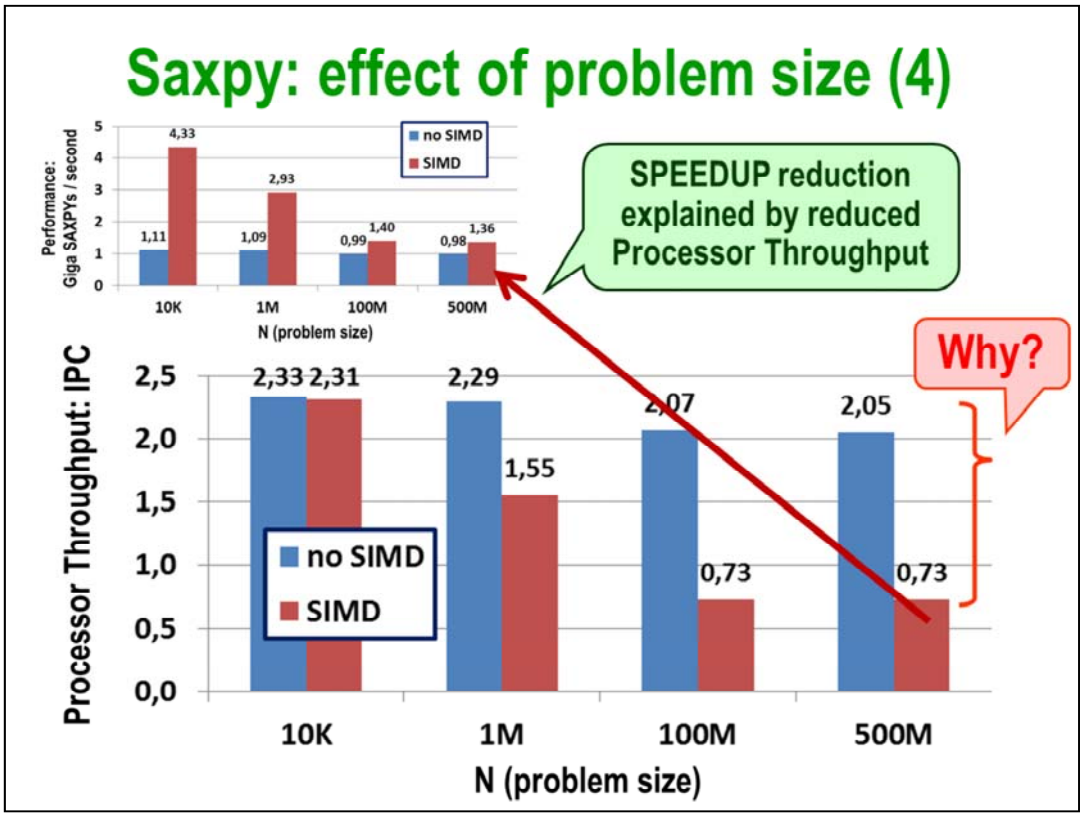


We now show the coding efficiency of the program, but measured as the number of machine instructions needed to encode a SAXPY operation. This result is obtained by dividing the total number of machine instructions executed between the total number of SAXPY operations performed, i.e. N. We can extract two conclusions from the data of the previous graph in a very intuitive and easy way.

First, the coding efficiency of the program does not change (or barely changes) with the size of the problem, or at least for the range of values from N = 10K to 500M. The small variation may simply be due to small errors in the measurements that are made during the execution.

The second and most important conclusion is that the vectorization of the code (the use of SIMD instructions) improves the coding efficiency with respect to the execution without SIMD instructions (or scalar execution). This makes perfect sense since SIMD instructions do 4 times more work than scalar instructions: the program can perform the same amount of SAXPY operations by executing four times less machine instructions if the instructions are SIMD instead of scalar.

As this factor improves 4 times in the vectorized code, we can explain the performance improvement of 4 times in the program for N = 10K (see the arrow shown on the slide). However, this factor alone does not explain what happens when the problem size grows: why despite running 4 times less instructions the program does not run 4 times faster, when N is greater than 1M?



We now show the IPC of the execution, or average number of machine instructions executed per clock cycle. This result is obtained by dividing the total number of executed machine instructions by the total number of "consumed" clock cycles to execute these instructions. The IPC gives us a measure of the utilization of the computational capacity of the processor. For example, if the maximum theoretical execution capacity is 4 instructions per cycle, the graph shows that the obtained empirical values are between 2.33 and 0.73, depending on the size of the problem and the version of the program. Again, we can extract two conclusions from the data on the previous graph in a very intuitive and easy way.

First, with a size of N= 10K, the IPC obtained is acceptable: 2.33 / 4 = 58.25%. As each SAXPY operation is 7 instructions in the scalar version, the IPC of 2.33 implies a time of 7 / 2.33 = 3 clock cycles per SAXPY operation. In the case of the SIMD version there would also be 3 clock cycles but every 4 SAXPY operations. In 3 cycles, 3 MEM μoperations, 2 FLOAT μoperations, 2 INT μoperations and one BRN μoperation are executed. It does not seem like a bad result.

On the other hand, by increasing the size of N, the IPC decreases, especially in the case of SIMD execution. The explanation is that the data of the vectors takes more time in being read of memory when the vector is great than when it is small. In a few slides we will explain why.

Finally, it is possible to explain why the problem of accessing memory affects more the SIMD version than the scalar version. Briefly, when the data read from memory arrives fast, the execution is limited by the time required to compute with this data (compute bound), and in this aspect the SIMD version is quite superior; On the other hand, when the data read from memory comes slowly, the two versions of the program are limited to execution at the rate of memory (memory bound), and in this case the rhythm of execution is similar. Sometimes an analogy regarding some familiar scenario helps understanding the same problem in an unknown or not well-known scenario. If we compare the speed of a car and a bicycle on a straight and well paved road, then we can find the potential difference of the vehicles in the best scenario for both of them. However, if we put both vehicles running in a road with many curves and poorly paved, then the potential of both vehicles equals. In other words, a curvy and poorly paved road degrades the performance of the car relatively more than the performance of the bicycle.

Example: 2-D Matrix Reduction

```
#define N 40000
double f ( double X[N][N] )
{
  int i, j;
  double S=0.0;
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      S= S + X[i][j];
  return S;
}
```

Elapsed Time: 3.268 seconds

Encoding Efficiency = 4.02 instructions / operation

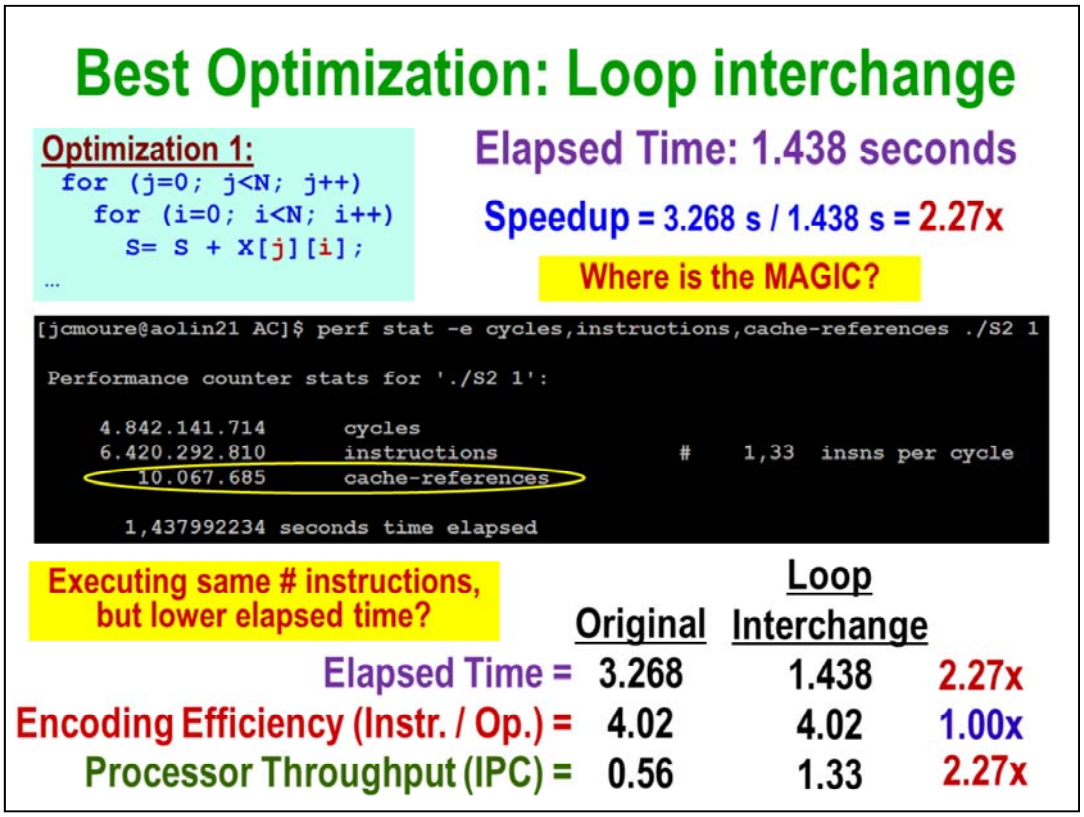
Processor Throughput (IPC) = 0.59

```
[jcmoure@aolin21 AC]$ gcc -O2 sampleALL.c -DREP=1 -o S2
[jcmoure@aolin21 AC]$ perf stat -e cycles,instructions,cache-references ./S2 0

 Performance counter stats for './S2 0':

    10.834.036.670      cycles
     6.426.009.278      instructions              #    0,59  insns per cycle
     1.605.965.018      cache-references

       3,268091638 seconds time elapsed
```

In this other example, a bad programming practice is shown, which can only be understood if the operation of the cache memory is understood.

The first step in detecting that the program is not efficient is to apply the Performance Engineering strategies we have been describing. Processor H/W counters, which measure performance events, provide us with information to make a diagnosis. Using the perf command, we measure these events, namely the clock cycle (ClockCount, or CCount) and the total number of instructions executed (InstructionCount, or ICount) and calculate the metrics we call Encoding Efficiency or machine instructions executed by program operation) and IPC or Processor Throughput (which would be the raw execution speed obtained by the program, in terms close to the H/W). The coding efficiency of 4 machine instructions per operation seemed good to us at the beginning of the course, although now that you know about the SIMD execution capacity we should no longer consider it good. On the other hand, knowing that the processor can reach a maximum IPC of 4, a simple analysis allows us to detect that the processor achieves a performance of 0.59 / 4 = 0.147, about 15% of the maximum execution capacity, below expected.

Now is the time to look at another H/W counter, which measures events called "cache references". We measured a value of about 1.6 billion "cache references". We do not yet have criteria to consider if the value is very large or not, but if we normalize this value to the total number of executed instructions we would get a cache reference every 6.400 / 1.600 = 4 instructions executed. We will see later that each of these events has a "cost" of tens of clock cycles, and that therefore a proportion of one event every 4 instructions is a considerable problem.



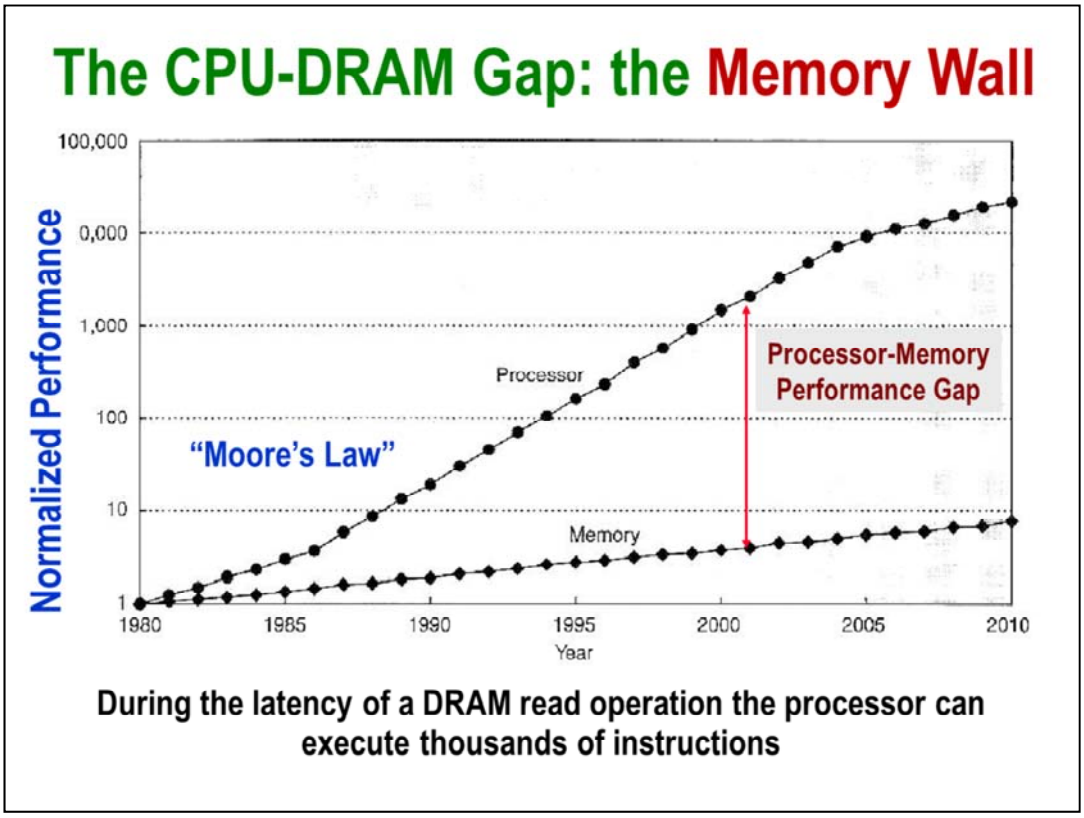
Modifying the program with a change of order between array indexes (optimization known as "loop interchange"), results in an (unexpected?) performance improvement of 2.27 times. As the coding efficiency hardly varies, the explanation for the improvement must be sought in the best processor performance, with an IPC that improves from 0.56 to 1.33.

But why going through matrix indices in a different way leads to better utilization of the processor's computing capacity? Because the processor use a memory hierarchy that includes caches: the detailed explanation will have you a little later. Note that the number of hits to the last level cache has decreased a lot (as you can see in the slide data highlighted in yellow).

Returning to Performance Engineering, the observed that the gain of the IPC coincides with a reduction of the "cache references" event to one every 642 executed instructions. We can correlate one thing with the other: reducing this event is the cause of the improvement in program performance. We will see in this subject that yes, that the change in the program (exchange indexes) modifies the trace of accesses to memory, which in turn reduces the occurrence of cache-references, and finally benefits the performance.

When a programmer "knows", an optimization can be as simple as exchanging the indices in the inner loop statement of the program. The goal of this topic is for students to understand why this simple modification improves performance. This involves understanding the operation of the cache with a global, high-level vision, not just a local and purely mechanical view (and that will be the first description we make on this issue).

Ready to get started?



The performance of a memory storage element has two different dimensions: (1) access latency and (2) memory bandwidth. Access latency is the time required to retrieve a datum from or write a datum to a given memory address (a memory address is an index identifying some memory location), measured in (nano) seconds or clock cycles. Memory bandwidth is the amount of data that can be retrieved from or written to memory in a unit of time, measured in (Giga or Mega) Bytes per second.

Compute performance can also be measured by execution time or execution throughput (programs or operations or machine instructions per second).

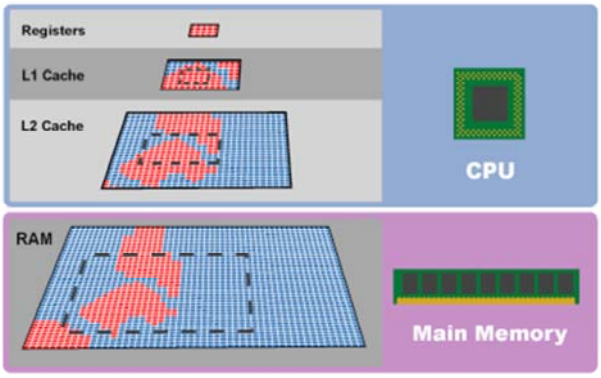
Along the last decades, when we compare the performance latency of compute and memory operations we find an increasing gap. This phenomenon is known as the Memory Wall. The final effect is that during the latency of a single DRAM read operation each new processor can execute an increasing number of instructions. The reason of the performance lag is not related with some inherent problem with the technology of memories, the problem is that, as processors are more powerful, memory capacity is also growing larger and larger: it is not possible to build memories that are both faster and with larger capacity.

Memory Technology

Small memory is faster than Large memory

SRAM: very fast memory technology (inside die)

DRAM: (off-chip)
- **more area-efficient & energy-efficient memory technology**
- **(but much slower)**

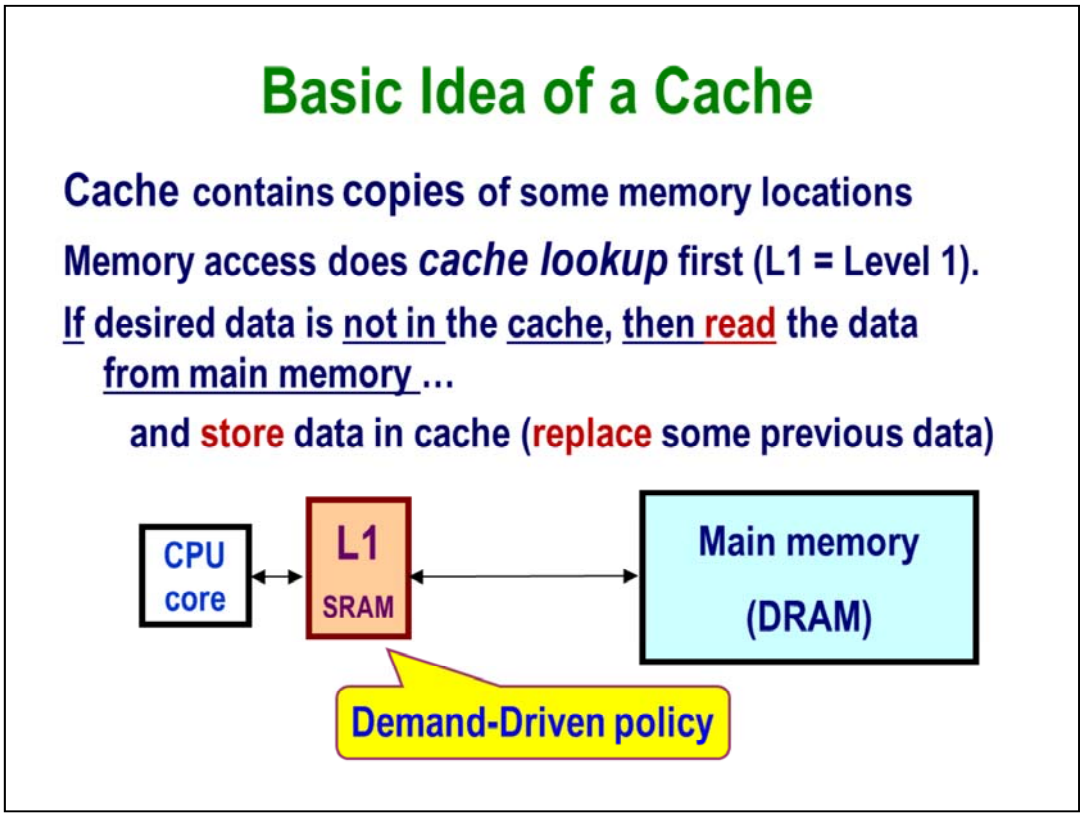


A small memory can be as fast as a compute unit, like an integer adder. This is the case for the internal registers of the computing core. The internal registers are a few number of storage elements placed very near to the units performing compute operations (additions, multiplies …). The number of registers is very small, between 16 and 128, depending on the processor. Since they are very few, machine instructions refer to them using a small set of numbers, like r4, or x13. In fact, reading from a register is extremely fast and most of the time is devoted to moving the data from the register file to the inputs of the compute unit and from the output of the compute unit to the register file.

A bigger memory can be constructed that takes only two or three times the latency of a single integer addition operation (2 or 3 clock cycles), but the capacity of this memory cannot be larger than 32-64 KiBytes. Memories with capacities of GiBytes have latencies on the order of 100s of clock cycles.

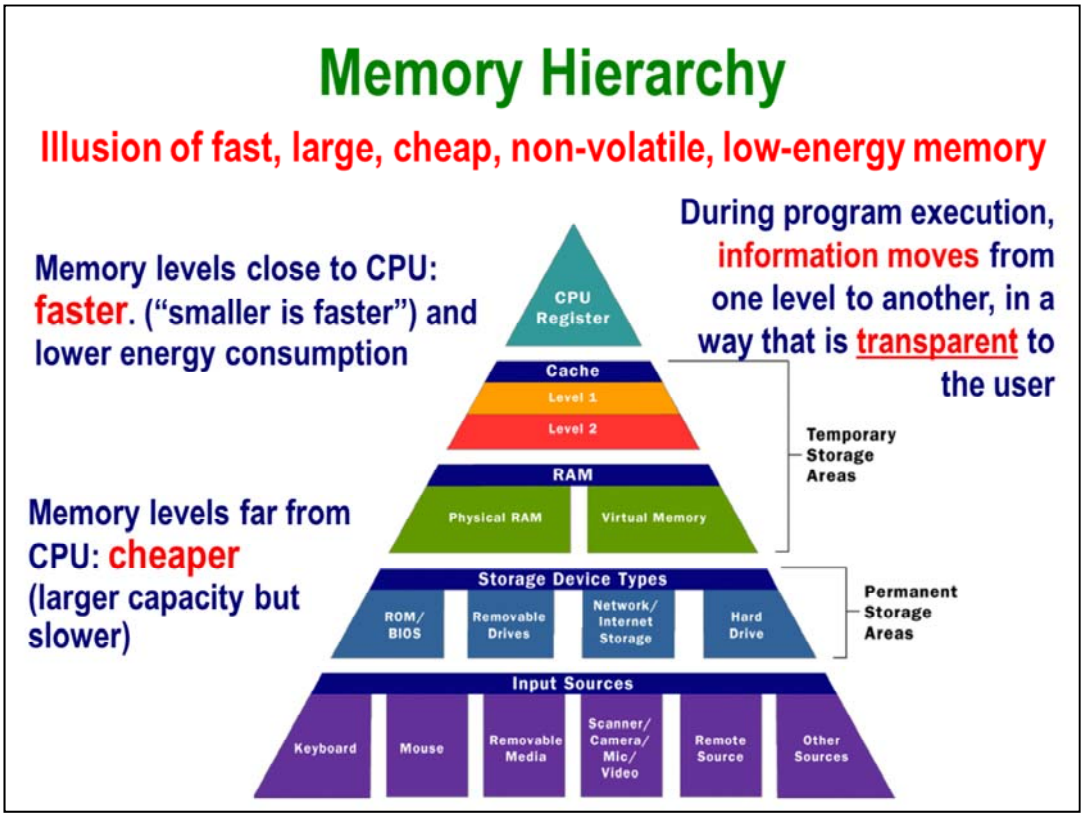
Static RAM (SRAM) devotes several transistors per memory cell and actively consume energy to maintain the storage contents. On the other hand, Dynamic RAM (DRAM) uses a capacitor for each memory cell, which only consumes energy when read o written, and when it must be periodically refreshed (around 50-100 times per second). SRAM access is faster than DRAM access, but DRAM memory consumes less energy and is cheaper to construct (much more memory cells per euro).

SRAM technology is used for the smaller memory inside the processor chip, prioritizing the reduction of access latency, while DRAM technology is used for the larger memory outside the processor chip, prioritizing cost and energy consumption.



We cannot have very fast storage for all the data in our program, but we can have very fast storage for a subset of the program's data. A cache memory is a fast (small) memory (built using SRAM technology) that holds a copy of some of the datum stored in the large main memory (built using DRAM technology). The contents of the cache memory changes dynamically, adapting to the runtime requests of the program: the cache basically contains the last data accessed by the program. Newly accessed data replaces the data accessed some time ago. This is called a demand-driven data allocation policy.

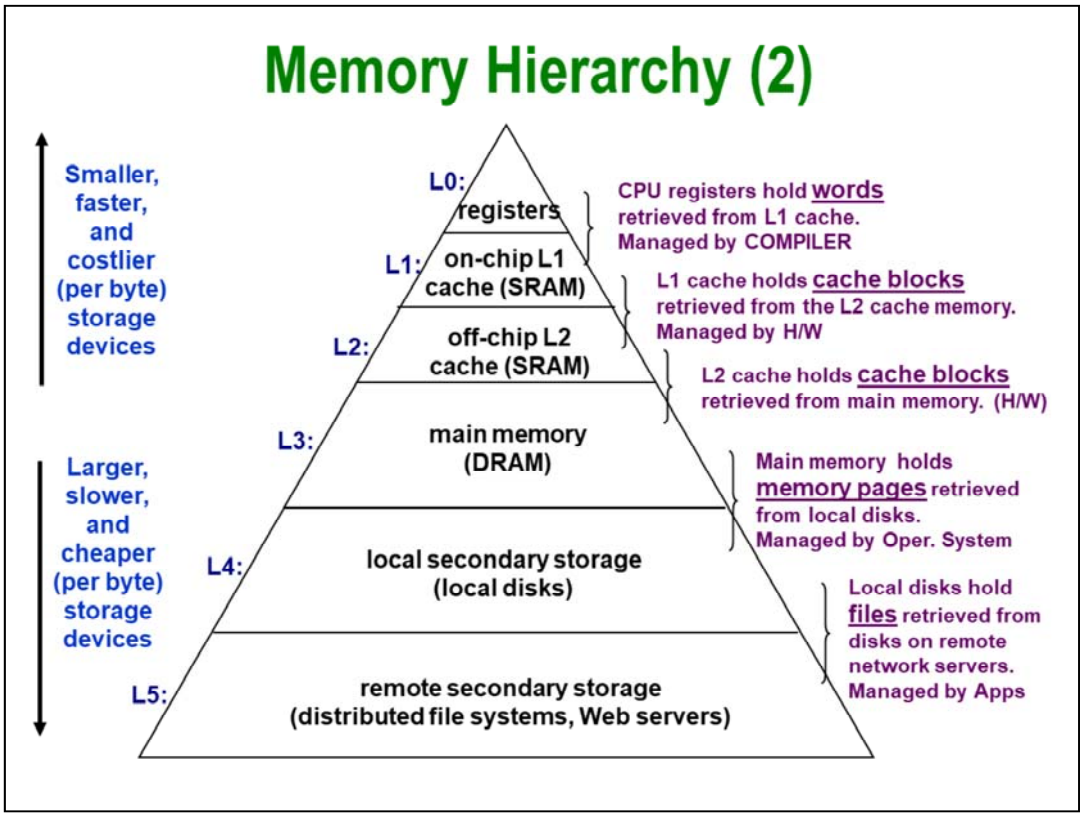
When a processor requests data from memory, it is first searched in the data cache: if it is found, then the data is provided to the processor very fast; otherwise the data is requested to the main memory, often with a long latency delay. Data coming from main memory replaces the data already in the cache.



Current systems contain more than one cache memory organized as a hierarchy. The cache on level 1 (L1 cache) is the fastest (and smallest, around 32 KBytes) cache, and the first place where the processor looks for a datum. Then the cache on level 2 (L2 cache) is the second fastest (and around 10 times bigger). Current computers usually contain a level-3 (L3) cache of several MBytes of storage capacity (even a L4 cache). The existence of the cache memory hierarchy is transparent to the programmer, even at the machine language level. Load and store instructions reference general memory addresses in a large memory space. Data moves automatically from one level of the cache hierarchy to another level of the cache hierarchy as a result of the load and store instructions executed by the program.

The existence of a cache hierarchy, though, is very important from a performance point of view. When a program accesses a small subset of data repeatedly, most of the accesses will hit on a fast cache, the latency of the memory accesses will be low, the IPC rate will be high, and the program's execution will be fast. Otherwise, if a program accesses a large amount of different data addresses, most or all of the accesses will miss in the cache and will be solved by the main memory, increasing the latency of load and store instructions, reducing the IPC rate and delaying program's execution.

The memory hierarchy spans to hard disks, remote storage servers … to provide the illusion of a very fast, very large, cheap, non-volatile (permanent) and low-energy consumption memory.



Each of the levels of the memory hierarchy contain data that is organized in different units of storage. On the lowest (and fastest) level of the hierarchy, each register contains a single word of 32 to 256 bits. Cache memories contain cache blocks of 64-128 Bytes (multiple words). When data is moved from memory to cache, a whole cache block is moved, the one containing the word requested by the processor.

Main memory is organized as a large set of memory pages (with a size that may range from 4 KBytes to a few MBytes): the virtual memory implemented by the Operating System in collaboration with the processor hardware moves memory pages between disk and main memory to provide the illusion that the memory of the computer is much larger than the physical DRAM memory actually being attached to the processor.

Finally, the unit of information moved between disks and remote servers is a variable-size file. However, distributed file systems often create fixed-size storage units (on the order of 10s or 100s of MBytes) to handle the movement of data between the processors attached to the distributed system.

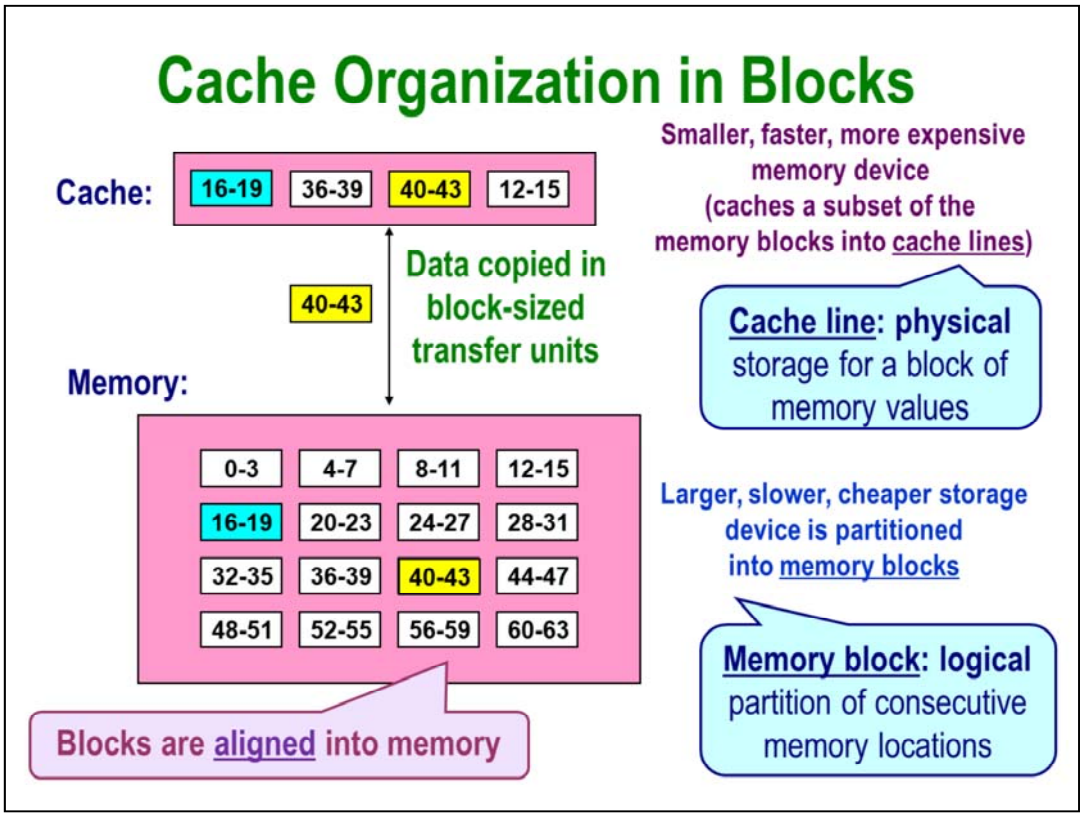
Processor: Intel i7 950 Nehalem (Lab)

Sandra 2009	Nehalem 3.2GHz	Penryn 3.2GHz
L1 Cache Latency	4 cycles	3 cycles
L2 Cache Latency	11 cycles	18 cycles
L3 Cache Latency	52 cycles	-

http://ark.intel.com/es-es/products/37150/Intel-Core-i7-950-Processor-8M-Cache-3_06-GHz-4_80-GTs-Intel-QPI

The processor of the computers of the Laboratory of AC (yellow label!) is an i7 950 of architecture Nehalem. In the slide you can see that it contains 4 execution cores, each of which can execute a program (or thread) independently (in fact, each kernel can execute two programs - or threads - at the same time, but this we will see much later).

You can also read the sizes of the cache, whose structure and operation is also explained in the picture. There is a large Level-3 cache with storage for 8 MB of data that is shared by the 4 execution cores. Each execution core has two private caches, a Level-2 cache (256 KB of storage) and a Level-1 cache (32 KB for data and 32 KB for instructions).

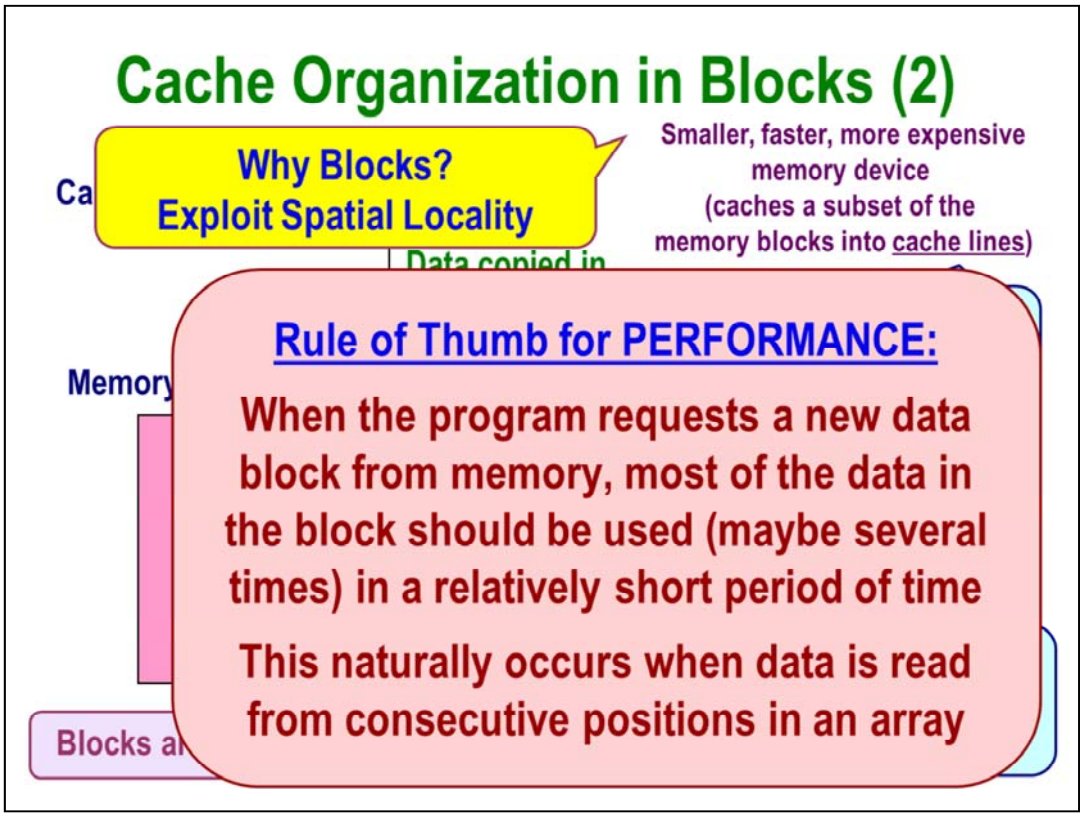


The example shows a cache memory capable of storing 4 data and a main memory capable of storing 16 data. The number associated with each datum that is drawn in each box represents the address of the data. The animation shows how the data in addresses 16 and 40 substitute the previous data that were cached. Remember: the cache memory can be fast, because it has small capacity. We can build it with expensive technology (both silicon and energy consumption), because the cost is cushioned by its small size compared to the size of main memory.

Although there is some (unsuccessful) processor design that does not meet it, we will assume the include property, that is, all the data that reside in cache are data copies that also reside in main memory. This scheme facilitates the management of the data between the caches. This slide shows that the data used by the program is organized into blocks of data: understanding this is fundamental to understand how the programmer can make efficient use of the memory system.

The program uses data ranging from one byte (char type) to 8 bytes (long type or double type). But it would be very complex to cache data of different sizes: to find out if a data item is or not it would be necessary to organize a complex structure in the form of lists, controlling the allocation of space within the cache as the C language runtime does / C ++ with malloc and free functions. Then, information in the cache memory is organized into fixed-size blocks.

Typical sizes for a cache block are 64 bytes or 128 bytes. The processor in the laboratory (intel i7 950) uses cache blocks of 64 Bytes. The cache contains space to store a certain number of blocks: each slot in the cache where a block can be stored is called a cache line. The blocks are aligned in memory. If the blocks are 64 Bytes, then the first block starts at address 0, the second at address 64, then the 3rd at address 128 ... Data moved between the cache memory and the main memory is always moved in blocks. Therefore, when the program requests a certain value from memory (for example a float value of 4 Bytes) and this memory location is not hold by the cache memory, then the main memory must provide the entire block containing the requested data (16 float elements).



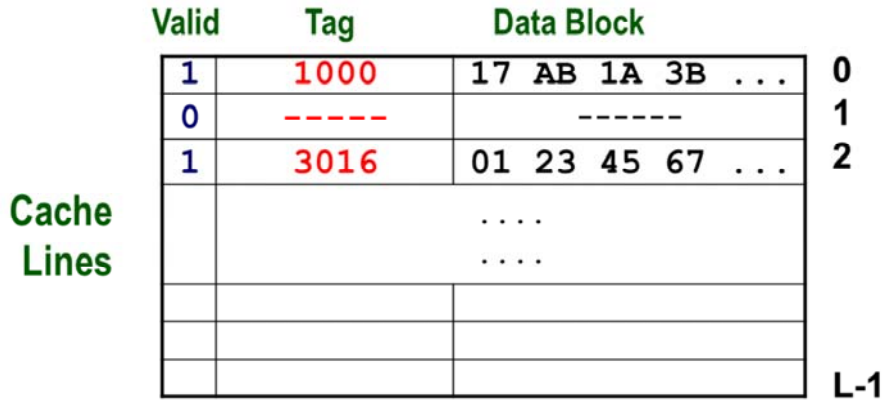
If data were misaligned, for example a long int of 8 Bytes in which the first 4 Bytes belong to block i, and the second 4 Bytes belong to the block i + 1, then it would be necessary for the compiler to generate a read instruction of 8 bytes with non-aligned access, or read the data using two 4-byte read instructions with which to "rebuild" the 8-byte data. Access to misaligned data reduces execution performance, either because more instructions need to be executed, or because the H/W needs more steps and increases access latency.

If a program accesses data by columns, then a different cache block will be requested for each data load operation. On the contrary, accessing data by rows means that the 16 data elements in a cache block are used in successive data load operations. In general, programs should try to access data in consecutive locations in the memory address space in order to improve the performance of the memory system.

The IPC improvement provided by the loop interchange optimization was explained by arguing that accessing data elements that are contiguous in the memory address space is more efficient and faster than accessing data elements that are scattered in the memory address space. Here we have provided the information to understand why.

Is some datum placed in cache?

- **Labels** (*tags*) identifying cache contents:
  memory address of a data block
- **Valid Bit** for data blocks placed in cache

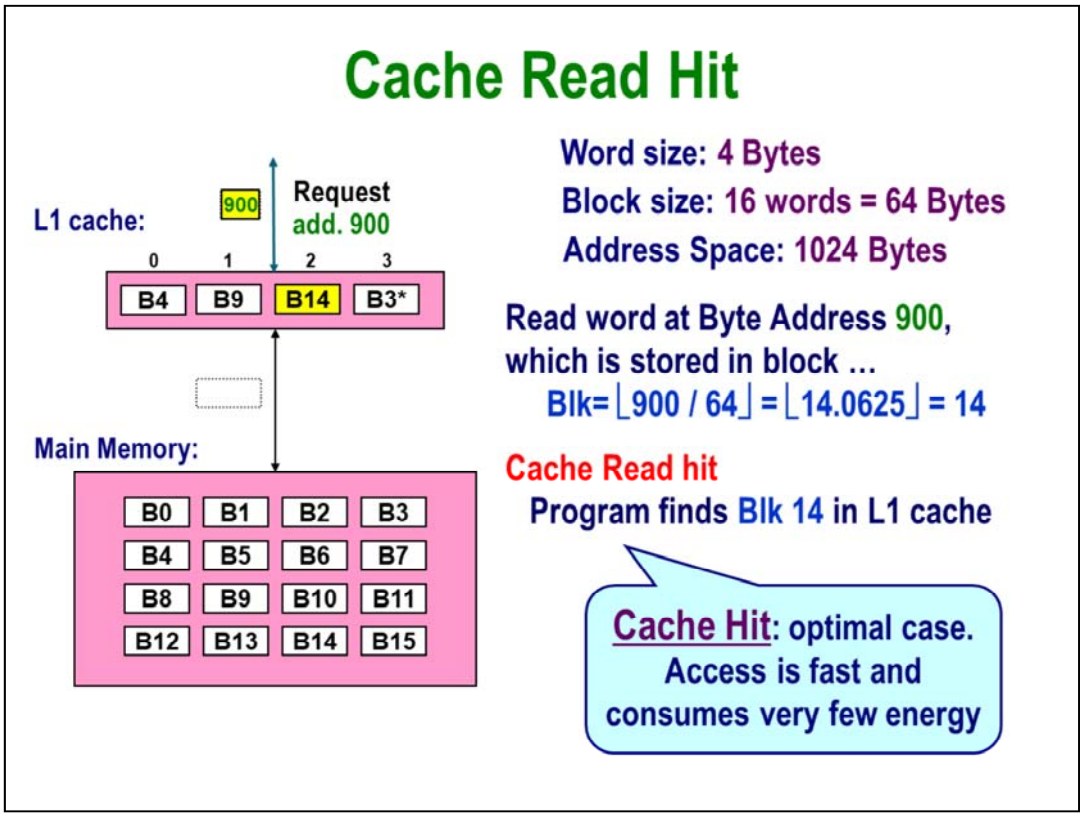


We will now explain how information is organized within the cache. Each cache line is divided into several fields: the validity bit, the tag and the data (in addition to other bits that provide additional meta-information on how to use the data, for example the bit that indicates whether or not the data has been modified within the cache).

The validity bit indicates whether the line information is valid or not. To delete the contents of a line is enough to deactivate its validity bit.

The tag identifies the data and specifies the address in main memory of the block it contains. Part of the address can be deduced from the line where the data is, so it is not necessary to store a complete address.

The data is always a fixed number of bytes (in our examples we assume 64). They are a copy of the main memory content, which is accessible much faster and energy-efficiently than the original.

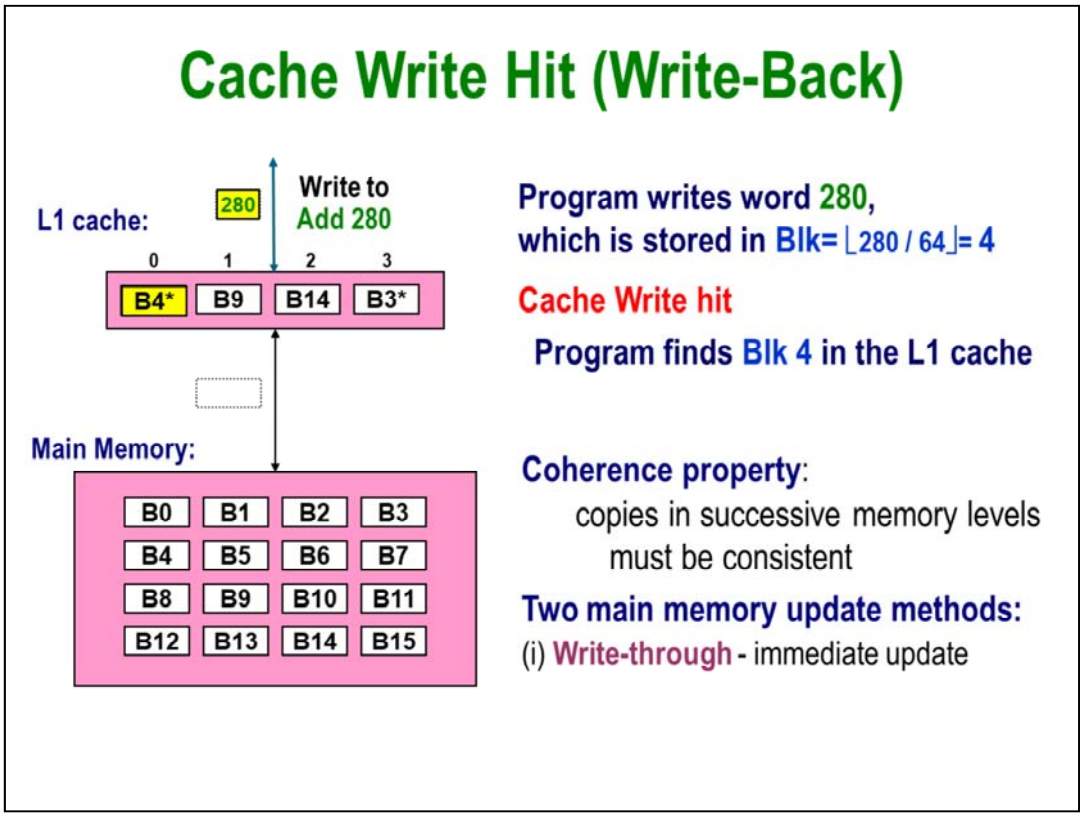


Now we will see how the memory address of a certain datum is converted in the identifier of the block containing the data. As an example we use a case of reading 4 Bytes at the address 900. We want to calculate in which block the data is, and then to search it in cache. As the blocks are 64 Bytes and are aligned, the block containing the data at address 900 is calculated by dividing the address by 64 and selecting the integer part: 900/64 = 14.0625, i.e., block 14.

Block 14 starts at address 14 * 64 = 896, and so the data we look for at address 900 is at the fifth position of the block, counting bytes.

The example shows that block 14 resides in cache, and therefore we say that there has been a hit in cache (cache hit). It is the desired case, because the access to the data is fast and with a small energy consumption. In order to check if the data is in cache, you must verify that there is a cache line that contains the tag that identifies block 14, and that the validity bit associated with the line is enabled. Once the line is identified, its contents are read, selecting the data within the part that contains the information.

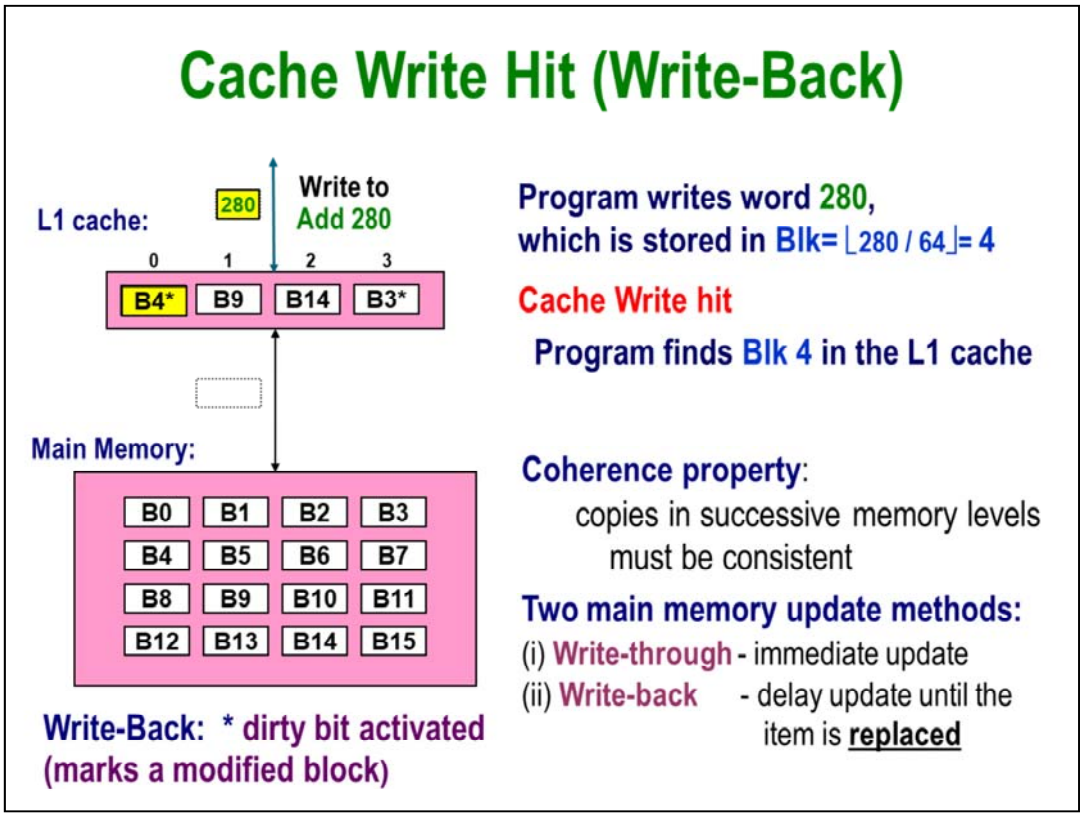
If the program reads block 14 multiple times in a short period of time (temporal locality), all accesses will be resolved quickly and with very low power consumption. In this way, the processors take advantage of the temporal locality of programs. The programmer should promote the reuse of the same variables (and of the same data) for short periods of time, to favor the good use of the memory hierarchy.



Write accesses require the same calculation to obtain the block identifier: the address 280 corresponds to block 4. The block is searched for in cache, and in this example it happens to be in cache (cache hit). Again, it is the desired case, because the writing of the data in cache is fast and saves energy.

The problem that appears is what to do with the content of the data that is still in memory. You have to maintain the consistency property that ensures that different copies of the same data within the system cache levels must be consistent, i.e. equal. The simplest strategy to ensure consistency is to immediately update the main memory with the new data value. This policy is called write-through, or immediate update. The update can be easily done in parallel and asynchronously to the operation of the processor: that is, the processor can continue executing instructions without having to wait for the data to be updated. If the program needs to read the same data in a short time, there will be a properly updated copy in cache, and will use it. The problem is that the energy consumption increases considerably for writes when data must always be written to memory. Another serious problem is that if the writing rate is high, the writing system can become saturated and end up slowing down the processor and considerably lengthening the execution time of the program.

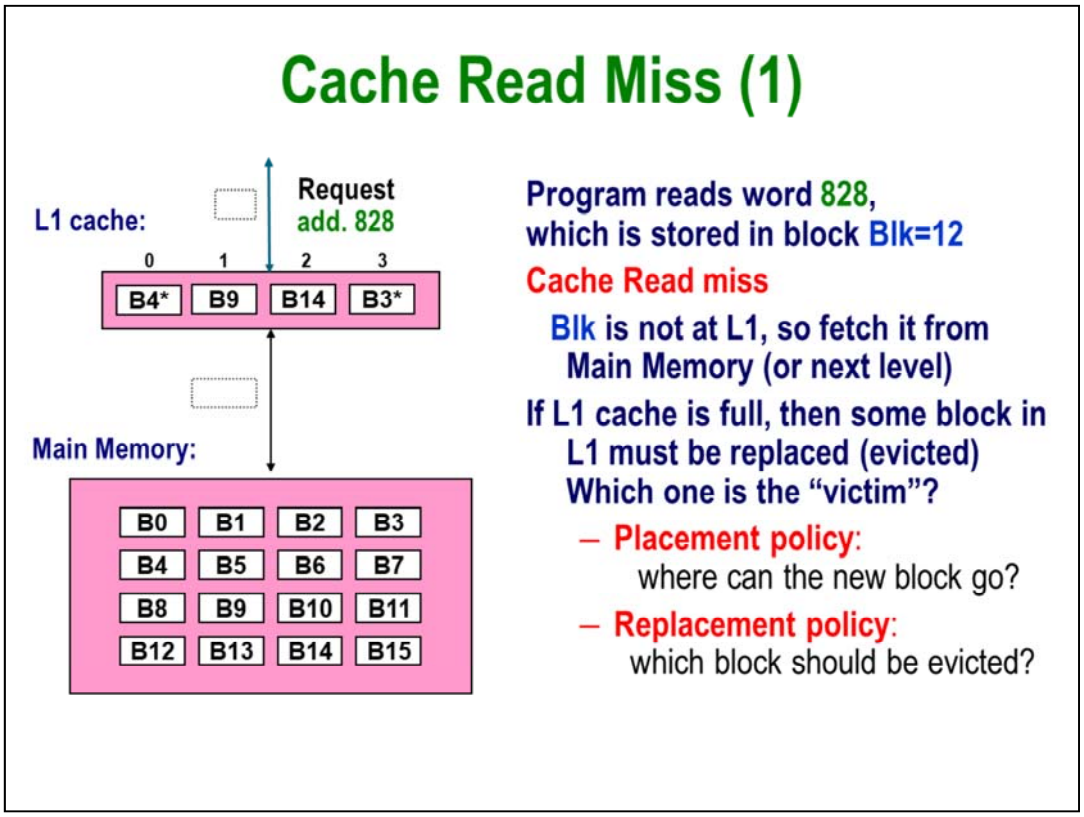
A strategy to improve the performance of the write-through policy is to "accumulate" the writings that are pending to be made in memories, and to gather them (what is called coalescing), so that writes to consecutive positions, for example writings of 4 Bytes to addresses 280, 284, 288, 292 ..., are converted to 16 or 32 or 64 Byte blocks of memory. In this way, less write operations are performed and they will be more efficient. If there are several modifications of the same data over a short period of time, some of the memory writes can be avoided.



Another more radical solution for cache writes is to use a policy called write-back, or delayed update, where you modify the copy that is cached but without updating the original data in main memory. A special bit ("dirty bit") is activated in the cache memory, indicating that the block of the cache line is modified with respect to the contents of the main memory. At the time the cache line space is needed to store new information, it is checked whether the modification bit is enabled. If so, the main memory is updated with the complete contents of the replaced block (because it is not recorded which bytes within the line have been modified and which have not been modified), and then the contents of the new block are cached.

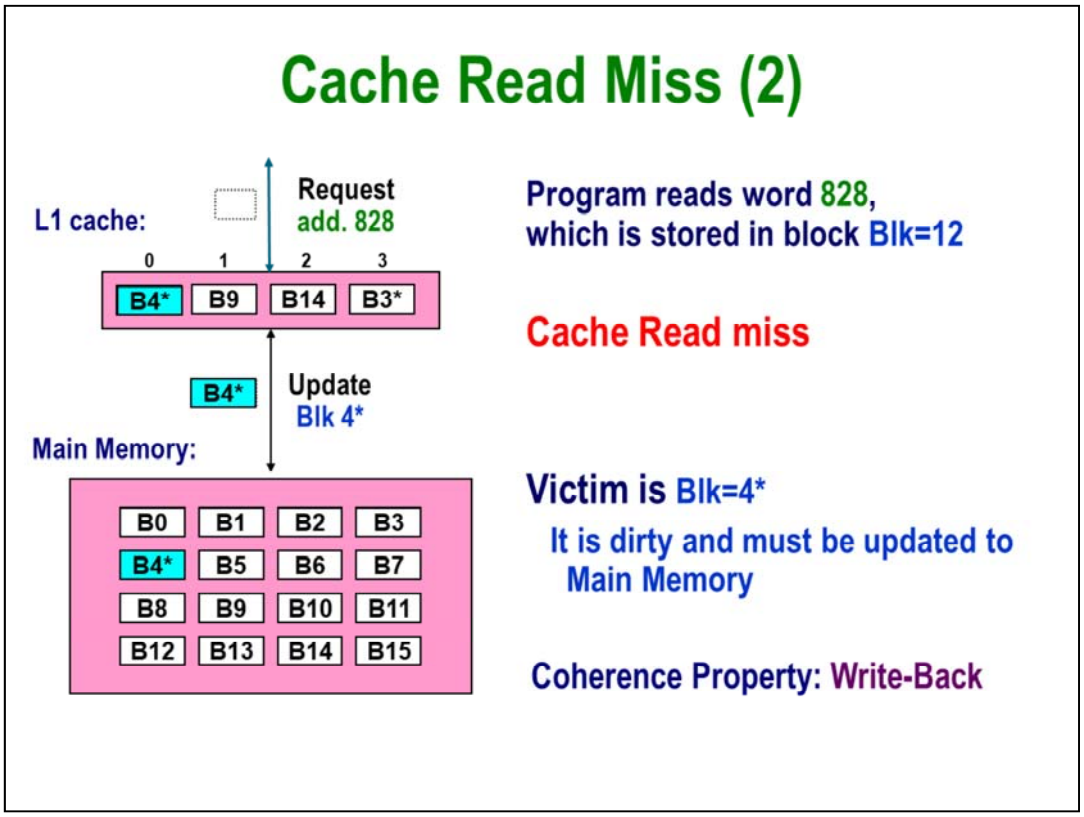
The advantage of this strategy, which is used by almost all processors, is that if the program makes many modifications to data that are in the same block for a relatively short period of time, all modifications will be made locally on cache memory (fast and efficient) and only an update in main memory will be made after the block is last modified and it turns out to be replaced by data of a different block.

Some processors offer instructions that allow the programmer to change this default policy. There must also be a strategy for deciding which blocks are replaced and where they are placed.



In the example, block 12 containing the data requested by the processor does not reside in cache, and therefore we say that a cache miss has occurred. This is not the desired case, because it means accessing the cache first and then accessing main memory, which is slow and with a high power consumption. The processor knows that the data is not cached by verifying that there is no cache line containing the tag that identifies block 12 with the validity bit enabled.

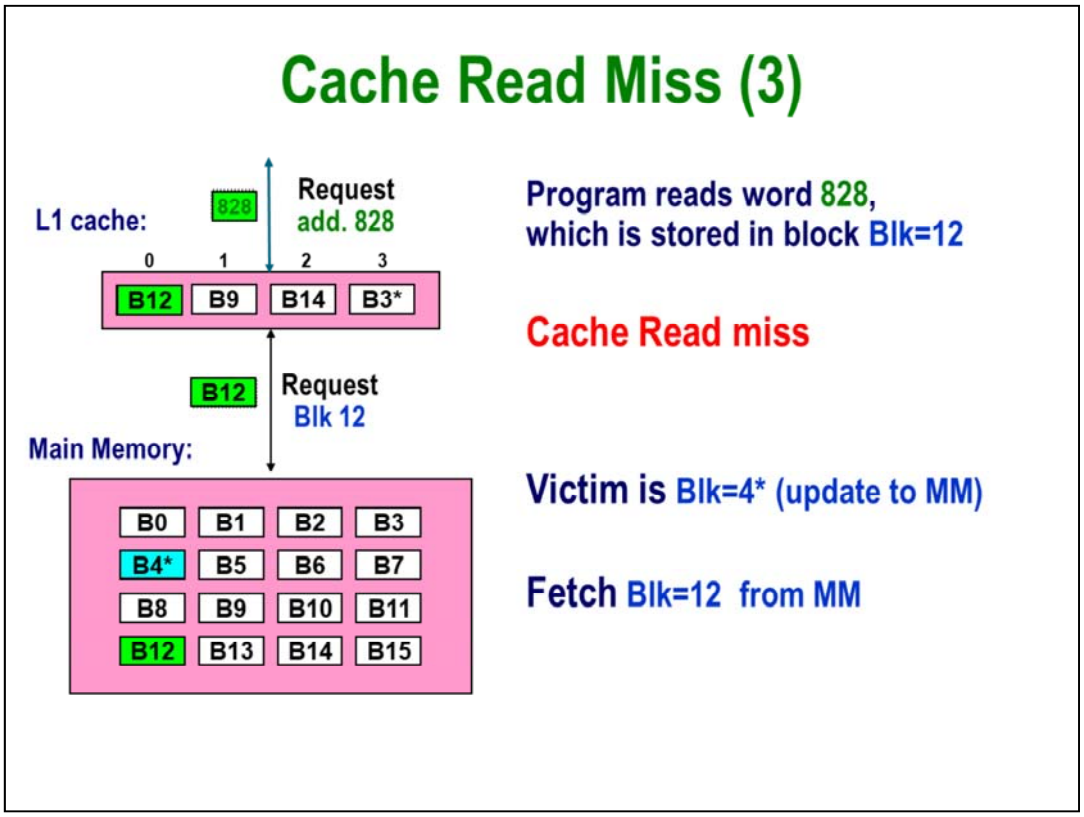
Once verified that the block that we requested is not cached, we select the line where the block is to be written, which is called the victim line. If the line has the modification bit enabled, it will be necessary to update the main memory with the complete contents of the replaced block. Then the new block is requested to main memory (or to the next level of cache, in case the system has multiple levels of cache), and is copied to the cache line. A policy is needed to decide which cache lines are likely to store the requested block (*allocation policy*), and which of these lines is best suited to replace the information it has with the new block (*replacement policy*).



In this example the victim is line 0, which contains block 4. It is not an arbitrary example, but represents a very simple type of policy: each block is located in the line that corresponds to its column. The column of blocks B0, B4, B8 and B12 corresponds to line 0, column 1-5-9-13 with line 1, etc. This policy is called direct-mapped.

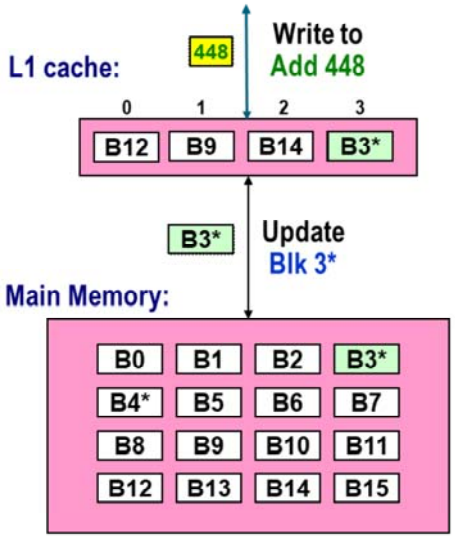
Important: The execution of a memory read instruction can cause, as in this case, both a main memory write and a main memory read.

To prevent block updates from having an impact on the performance of the program being executed, the processor is designed so that the accesses are reordered: the block to write to main memory is quickly written to an intermediate buffer, then the new block is read from memory and is cached, and finally the contents of the intermediate buffer (replaced block) are copied to main memory. Thus, the waiting time for the data requested by the program is greatly reduced.



Finally, block 12 is fetched from main memory to line 0 in the cache memory

Cache Write Miss: write-allocate (1)



Program writes word 448,
which is stored in block Blk=7

Cache Write miss

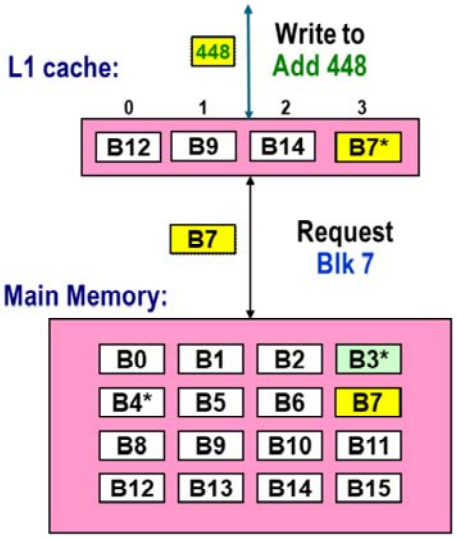
Blk is not at L1 cache, so …

Policy for write misses?

(i) Write-allocate – fetch and write

Write-Allocate: allocate a place for blk 7 by replacing blk 3

Cache Write Miss: write-allocate (2)



Program writes word **448**, which is stored in block **Blk=7**

Cache Write miss

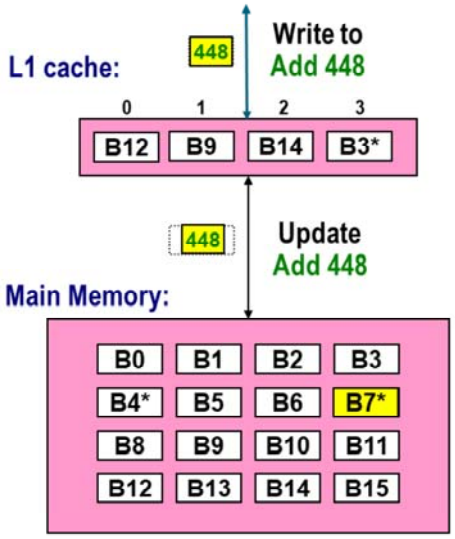
Blk is not at L1 cache, so …

Policy for write misses?

(i) **Write-allocate** – fetch and write

Write-Allocate: L1 cache fetches block 7 & updates it (set dirty bit)

Cache Write Miss: No-Write-Allocate



Program writes word **448**, which is stored in block **Blk=7**

Cache Write miss

Blk is not at L1 cache, so …

Policy for write misses?

(i) **Write-allocate** – fetch and write

(ii) **No-Write-allocate** – write to Main Memory but not allocate in Cache

No-Write-Allocate: L1 cache does not fetch block 7

Summary: Handling writes

Assure COHERENCE between Cache & Memory

Hits:

a) Update data both in cache and memory (*write-through*)

b) Write to cache only, update memory when block is replaced (*write-back*)

Misses:

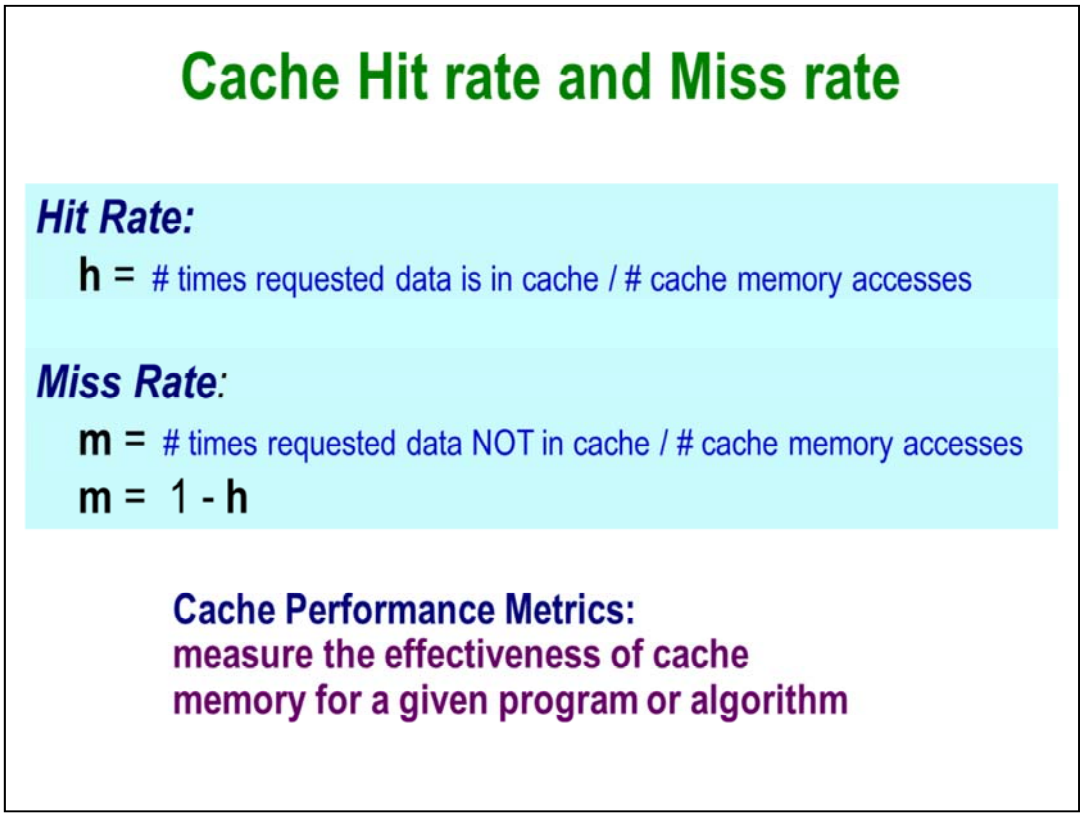
a) Move block to cache + update in cache (*write-allocate*)

b) Update on Memory, do not write on cache (*no-write-allocate*)

Examples:

a) Many writes per block: use *write-back* & *write-allocate*

b) A single write per block: use *no-write-allocate*

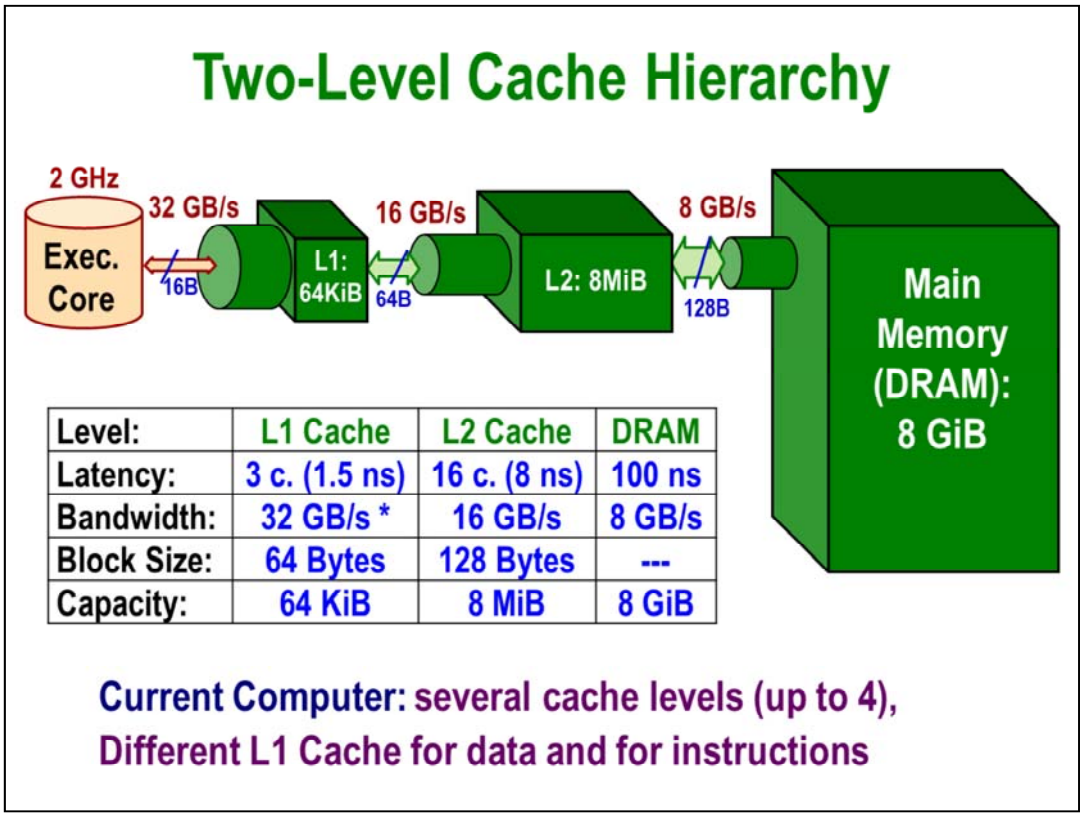


One way to measure the performance of the cache memory is to count the total number of misses that occur. The more misses, the worse, because they are likely to affect program execution, causing the processor to wait and slowing down the execution. Indicating the absolute number of misses is a partial form of reporting, since without further information we can not assess whether this number is high or low. In order to make this assessment better, it is preferable to use a relation between the total number of misses and the total number of accesses that the program performs. This is called the cache miss rate, and is a measure that depends on the execution of the program and, in a general case, difficult to estimate a priori. The cache hit rate is the complementary value, so the sum of the two rates must be one.

In a modern processor, the miss rate does not directly determine the penalty that occurs in the execution time of a program. The processor performs many tasks at once, and accessing the data is only one of them (but very important). The reordering mechanisms for the execution of instructions that the processor performs dynamically sometimes reduces the penalty of cache misses or even avoids that penalty. But, in general, a high miss rate indicates that we should investigate whether the effect of cache misses is a bottleneck for performance.

Sometimes the relative miss rate instead of the global miss rate is measured: for example, if we have a system with two levels of cache (L1 and L2) we can measure the miss rate in L2 as the relationship between misses in L2 and accesses to L2. This measure is dangerous because it can deceive us. For example, a rate of 1, that is, 100% of misses, may not be as serious as it seems if the total number of accesses to the L2 cache is very low, for example because more than 99% of the time the data are at the L1 level of cache and it is not necessary to request data at L2. As an analogy, it is not so bad to always lose in bingo if we only play once in our life.

Likewise, a high failure rate is not as severe if the program executes very few memory access instructions as compared to arithmetic and logical computation instructions.

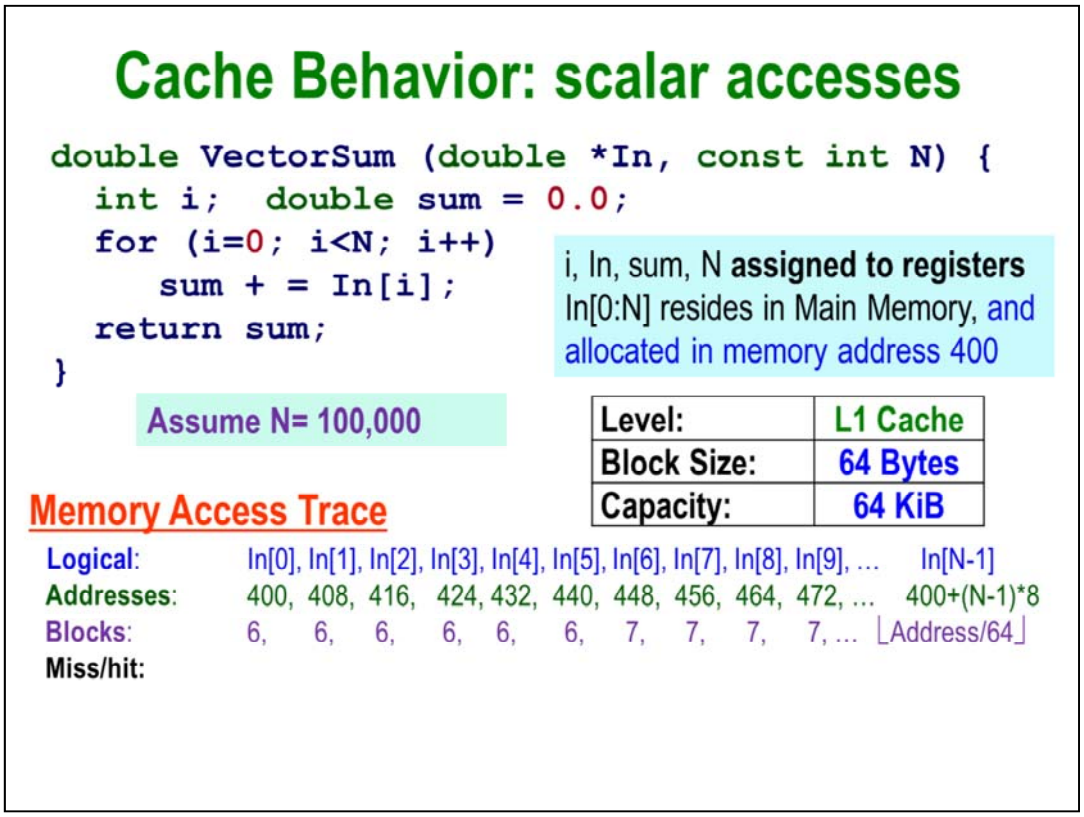


This slide shows the memory system that we are going to use as an example from now on. It is a system with 2 levels of cache memory. The execution core (which operates at a clock frequency of 2 GHz) requests data at the L1 level, which responds with a latency of 3 cycles (1.5 ns). There is a L1 cache specialized in storing data and an identical one specialized in storing instructions. Both have the capacity to store up to 64KiB (note that the i of KiB indicates that 1 KiB = 1024 and not 1000). The data is stored in blocks of 64 Bytes, and therefore L1 caches contain 1024 lines each. On the other hand, they are able to provide data to the execution core (and provide instructions to the front-end decoding instructions) at a maximum rate of 32 GB/s. This rate is called cache access bandwidth.

When misses occur in L1 cache, the requests are redirected to the L2-level unified cache. Unified means that it contains both data and instructions. If the searched data is in the L2 cache, then the entire 64 Byte block containing the data is moved to the appropriate L1 cache (data or instruction). If the data is not in L2 cache, the request is passed to DRAM. DRAM will reply with a block of 128 bytes, which is the size of the L2 cache blocks, and will be cached in L2. In addition, half of this 128-byte block containing the data requested by the execution core (a block of 64 contiguous Bytes) will be copied from L2 to L1. Finally the data will be sent from L1 to the execution core.

The L2 cache has a capacity of 8 MiB and, being larger than L1, the access latency grows to 8 ns (16 clock cycles) and the maximum bandwidth decreases to 16 GB/s. The decision that the L2 cache block is 128 Bytes has nothing to do with the total capacity of the L2 cache: it is a decision that the designers make based on the analysis of both the behavior of the programs and the cost of the H/W necessary to do so (in transistors, as in energy, as in time).

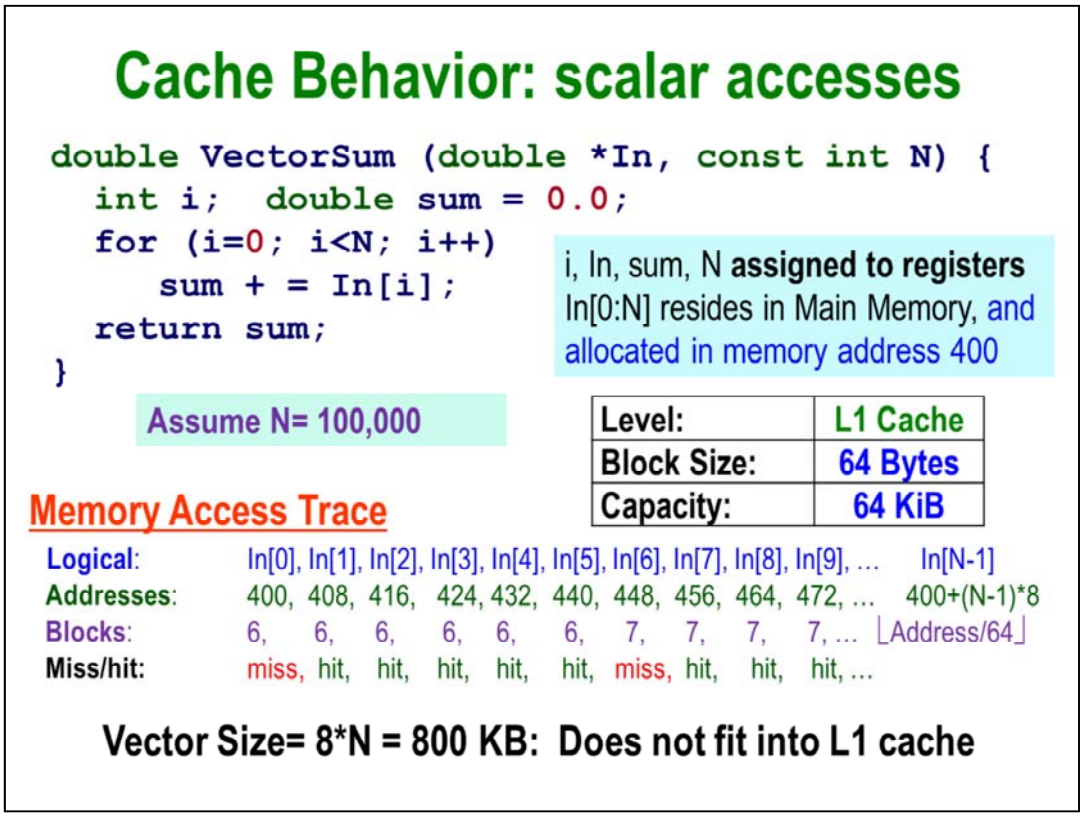
DRAM memory has a large storage capacity at the cost of very high latency and lower bandwidth. There is no point in talking about DRAM block size. In any case we should talk about page size, which is the unit that is transferred between the DRAM and the disk when the Operating System implements the mechanism of Virtual Memory.



Next we will analyze the operation of the memory system with a simple but real program. We first need to identify the memory access trace of the program's execution. The trace is the sequence of memory addresses that the execution core sends to the L1 level cache. We focus only on data access tracing, and ignore the trace of instructions, since the access to the instructions does not usually lead to poor processor performance. In order to identify the data access trace, it would be necessary to analyze the program's assembly code, but we can already make assumptions about how this code should be. The compiler will assign the i, In, N, and sum variables to processor registers, so all uses of these variables will not need to access memory. This is the best possible option, since the accesses to memory increase the total number of executed instructions, and these instructions are usually instructions of high latency. We can assume that the compiler will do its job very well, as long as the number of local variables is less than or equal to the total number of registers available to the processor.

Thus, only accesses to the elements of the vector In [] need to generate memory access instructions. The sequence of accesses is shown in the slide in a logical way, using the nomenclature with which we refer to these data in C language. We are assuming that the compiler does not generate a vectorized version (SIMD) of the code, although if so the results obtained would be similar. Now you have to convert this logical trace to the address trace that the program actually generates at runtime.

There are several ways of empirically discovering the direction of the first element of the vector In[] (printf, debugger, ...). In this example we will assume that this address is 400. Since double data occupies 8 Bytes, it is easy to generate the complete address map. The slide shows the first 10 addresses and the formula that gives us the last address. The reader is left to discover all the missing addresses ... The next step is to convert the addresses to block identifiers, something that we have already done just a few slides ago. Knowing the size of the block of the L1 cache (64 bytes) we can calculate the addresses to blocks of L1.



Now comes the really interesting work: to "mentally simulate" (or pointing it out on a paper, with drawings, ...) what will happen during the execution with this sequence of accesses. This way we can identify all accesses that cause cache misses.

As shown above, we assume that the vector In[] has N= 100,000 elements. Since the vector data is of type double, the vector occupies 800 KB (see slide with calculations, and remember that K= 1000, while Ki = 1024). Clearly, the vector does not fit into the L1 cache (with a capacity of 64 KiB). Therefore, it is reasonable to assume that the first blocks of the vector will not be cached. At most, it is possible that shortly before executing the above code the vector has been accessed, probably in the same order as in the program (from the element zero onwards, one at a time, and until the last element). In this case, only the last 64 KiB of the vector (about 8000 elements) would reside in cache.

It seems reasonable, therefore, that when executing this code there are no (or almost none) of the elements of the vector In [] contained in L1 cache. Therefore, the first access to block # 6 will miss, and will cause the next level of the memory hierarchy (L2 cache) to be read and L1 be cached. Thus, the following 5 accesses to block 6 will NOT cause cache miss. The program then requests block 7 to cache L1 and a cache miss will occur again, followed by 7 successive accesses to block 7 that will be successful (hits). During the rest of the run, cache misses will occur every 8 cache read requests. As blocks 6, 7, 8 ... are being replaced, the blocks that were cached will be replaced, and therefore, when the program requests the final blocks of the vector, these will no longer be in the cache.

Cache Behavior: SIMD accesses

```
double VectorSum (double *In, const int N) {
  int i;  double sum = 0.0;
  for (i=0; i<N; i++)
     sum + = In[i];
  return sum;
}
```

i, In, sum, N **assigned to registers**
In[0:N] resides in Main Memory, and allocated in memory address 400

Assume N= 100,000 & SIMD LOADs

Level:	L1 Cache
Block Size:	64 Bytes
Capacity:	64 KiB

Memory Access Trace

Logical:	In[0:2],	In[2:2],	In[4:2],	In[6:2],	In[8:2],	In[10:2], …	In[N-2:2]
Addresses:	400,	416,	432,	448,	464,	480, …	400+(N-2)*8
Blocks:	6,	6,	6,	7,	7,	7, …	⌊Address/64⌋
Miss/hit:	miss,	hit,	hit,	miss,	hit,	hit, …	

Vector Size= 8*N = 800 KB: Does not fit into L1 cache

The cache behavior when the program is vectorized is very similar, except that there will be one miss every 4 accesses, because now each access performs double work (reads 16 bytes instead of just 8 bytes).

Cache Behavior (2)

```
double VectorSum (double *In, const int N) {
  int i;  double sum = 0.0;
  for (i=0; i<N; i++)
     sum + = In[i];
  return sum;
}
```

Assume vector In[] has been just initialized

Level:	L1 Cache	L1 Cache	DRAM
Block Size:	64 Bytes	128 Bytes	---
Capacity:	64 KiB	8 MiB	8 GiB

Minimum Number of Cache Misses depending on N:

N	Size	# L1 misses	# L2 misses
5,000	40 KBytes	0	0
50,000	400 KBytes	400 KB / 64 B = 6,250	0
5,000,000	40 MBytes	40 MB / 64 B = 625,000	40 MB / 128 B = 312,500

The table below summarizes the behavior of the system in three cases with different vector sizes (values of N). We assume that vector In[] has already been written or read previously by the program and, therefore, if it fits in a level of cache we assume that the data will still be there. The size of a cache level determines whether the vector fits or does not fit. Therefore, for N= 5000 there are no L1 cache misses because the vector fits (40 KB). For N= 50,000, the 400 KB vector does not fit in L1 but in L2 (8 MiB capacity). We will ignore cases of vector sizes that are in the limit, because in these cases, when more than half of the vector fits, the number of misses is smaller than if the vector is much larger than the cache.

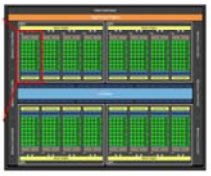
The size of the block (in the example, 64 Bytes) determines how many misses occur in total: there will be as many misses as different blocks occupy the vector. In the slide, the number of misses (equal to the number of blocks) is calculated by dividing the size of the vector by the size of the block. You can also calculate the number of data that will fit in a cache line (dividing the block size by the size of the data: in this example 8 data will fit in each 64 Byte block), and calculate the miss rate. Multiplying the total number of accesses (N) by the miss rate (1/8) we obtain the total of misses.

As an analogy to understand the operation of the cache with this program, we could think of thousands of yogurts organized in packs of 8: we are eating the yogurts consecutively, and when we finish a pack we continue with the next. Starting a new pack is equivalent to a cache miss, there are as many misses as packs, and you can calculate the number of packs (cache misses) by dividing the total number of yogurts by the total number of yogurts in the pack.

For N= 5,000,000, the 40 MB vector does not fit into L2. As the size of the L2 cache block is 128 Bytes, twice the size of the L1 cache block, L2 cache will produce half misses than in L1 cache: each miss in L2 assumes two misses in L1. The miss rate that occurs in the L2 cache when the vector does not fit is 1/16, while in the L1 cache with blocks of 64 Bytes the miss rate when the vector does not fit is 1/8.

Important: this analysis of the number of misses generated by the program is only valid for a program that accesses the data consecutively in memory. For this case, the larger the block size, the better, because fewer misses will occur. Put another way, for this program that exhibits maximum spatial locality, using large blocks is beneficial to exploit this feature of program behavior.

Processor Architecture

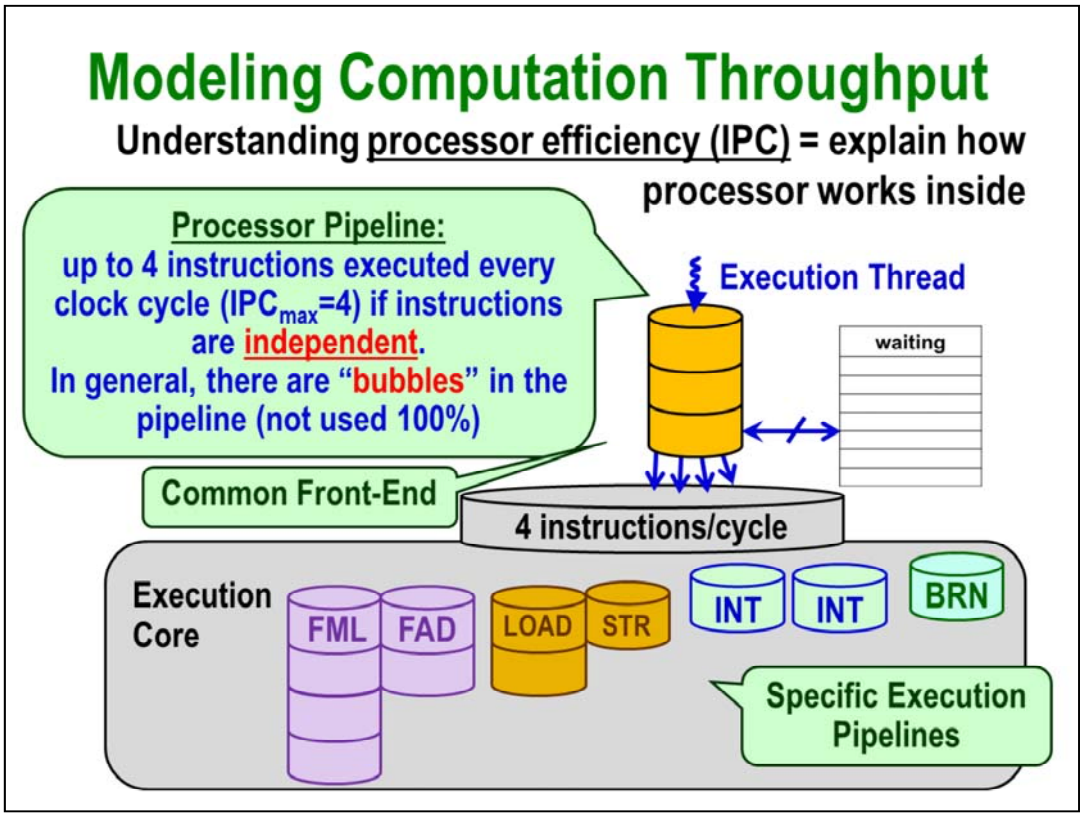


Processor Details to understand Performance

ILP (Instruction Level Parallelism):
how to execute multiple instructions in parallel

A computer core is able to execute several instructions at the same time, or in parallel. We say that the processor exploits the *Instruction Level Parallelism* (ILP) of programs; i.e. the existence of independent instructions in the sequence of instructions that a program wants to execute.

Next we will present a simplified model of a computer core, and then we will describe the two main problems that prevent achieving all the potential performance available in a computation core, namely (1) a unbalanced mix of instruction types, and (2) the existence of excessive data dependences among the instructions.

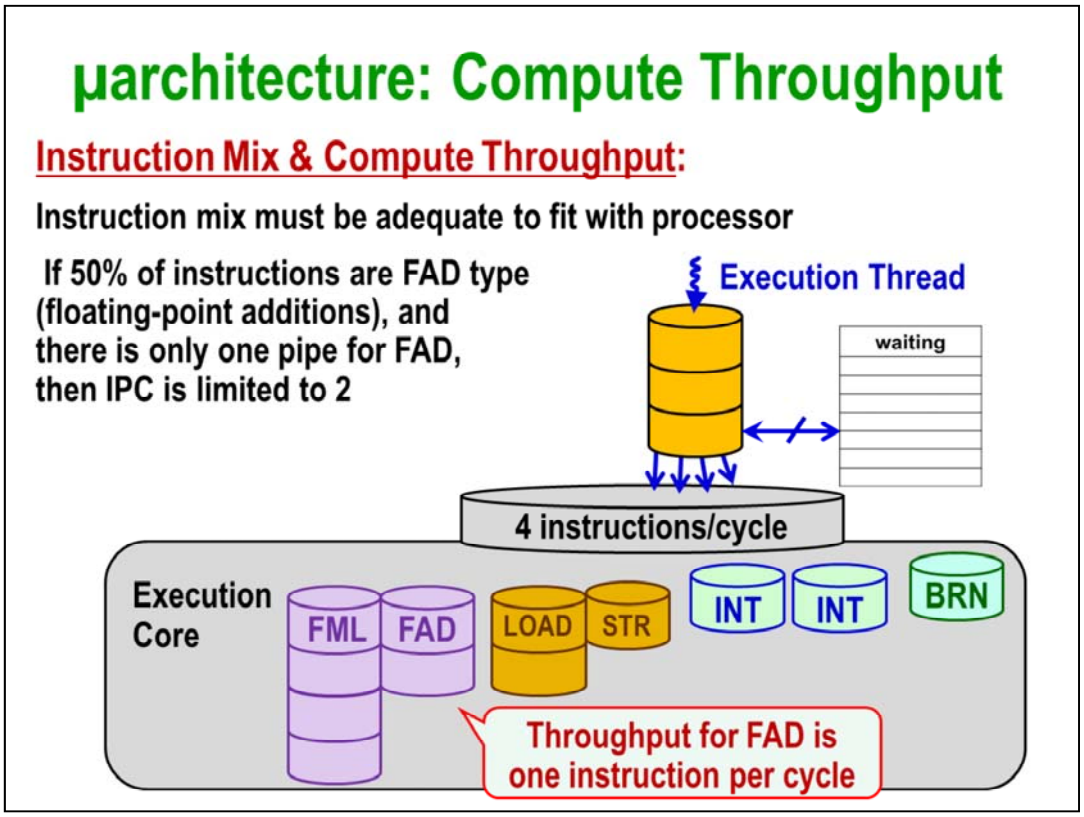


Is it reasonable that the execution achieves an IPC of, let's say, 0.9 (instructions executed per clock cycle)? In order to answer this question, we need to understand the basic processor microarchitecture, i.e., the way a processor works inside.

Current processors are able to execute several instructions per cycle, as long as instructions (1) do not depend on each other (are independent) and (2) do not claim the same execution and memory resources simultaneously. The execution of each single machine instruction is performed as a chain of sub-operations organized like a sequential pipeline. This organization of the execution of instructions allows several instructions to ENTER the computing pipeline simultaneously and many more instructions to BE TRAVERSING some stage of the processor pipeline at the same time. A good analogy of how a computer works is a car washing system composed of several car washing tunnels (the CPU pipelines), and each tunnel being able to contain several cars simultaneously, each car in a different stage of the washing process.

Therefore, a processor works like an assembly of pipelines: instructions enter the processor from a "main entrance" (the common pipeline, in the slide) at a certain maximum rate (4 instructions per cycle in the example of the slide and in the laboratory's processor), and then instructions are dispatched to specific computation pipelines. Each specific pipeline is devoted to execute a certain type of instructions. For example, the instructions classified as floating-point additions are sent to an specific pipeline that is labelled FAD. Some types of instructions may be executed in more than one pipeline (for example, there are two pipelines for general integer instructions, that are labelled INT).

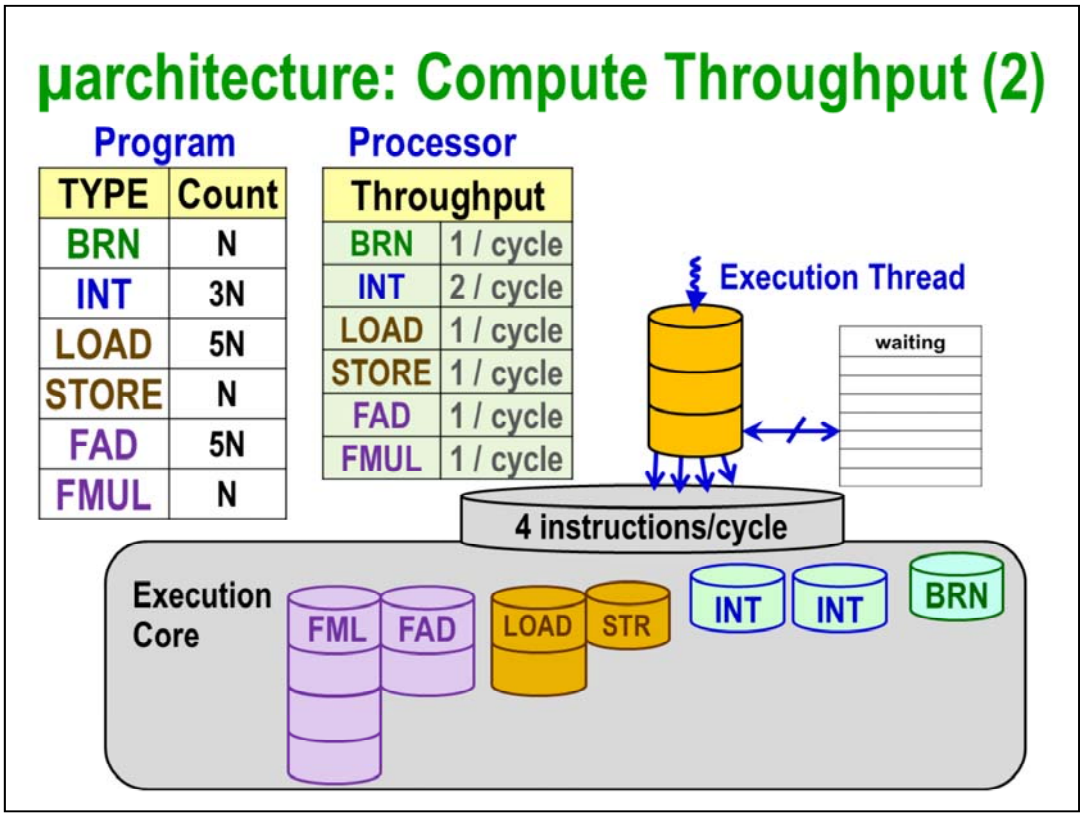
Instructions that cannot be executed immediately are stored into buffers or queues and wait there until they can get a pipeline resource and their input operands. The size of these buffers or queues is finite, and when too many instructions are waiting the pipeline entrance is becomes blocked, the processor compute capability is not fully utilized, and the efficiency of the execution drops (the IPC is reduced). In such a case, it is often said that empty bubbles are introduced in the execution pipeline. Performance is improved by reducing the amount of bubbles.



Not every combination of instructions can be executed simultaneously. This limitation creates bubbles in the pipeline execution. We call these problems resource stalls.

For example, the figure in the slide represents a processor that can execute up to 4 instructions every cycle (like the processors in the laboratory). The type of the instructions determine the actual compute resources that instructions will use. Then, for example, only two INTeger (INT) instructions can be executed at the same time, and only one Float MuLtiply (FML), one Float ADdition (FAD), one memory LOAD, one memory SToRe (STR) and one BRaNch (BRN) instruction can be executed simultaneously. I.e., the combinations of 4 instructions cannot include more than 2 INT instructions, or more than one instruction of any other type.

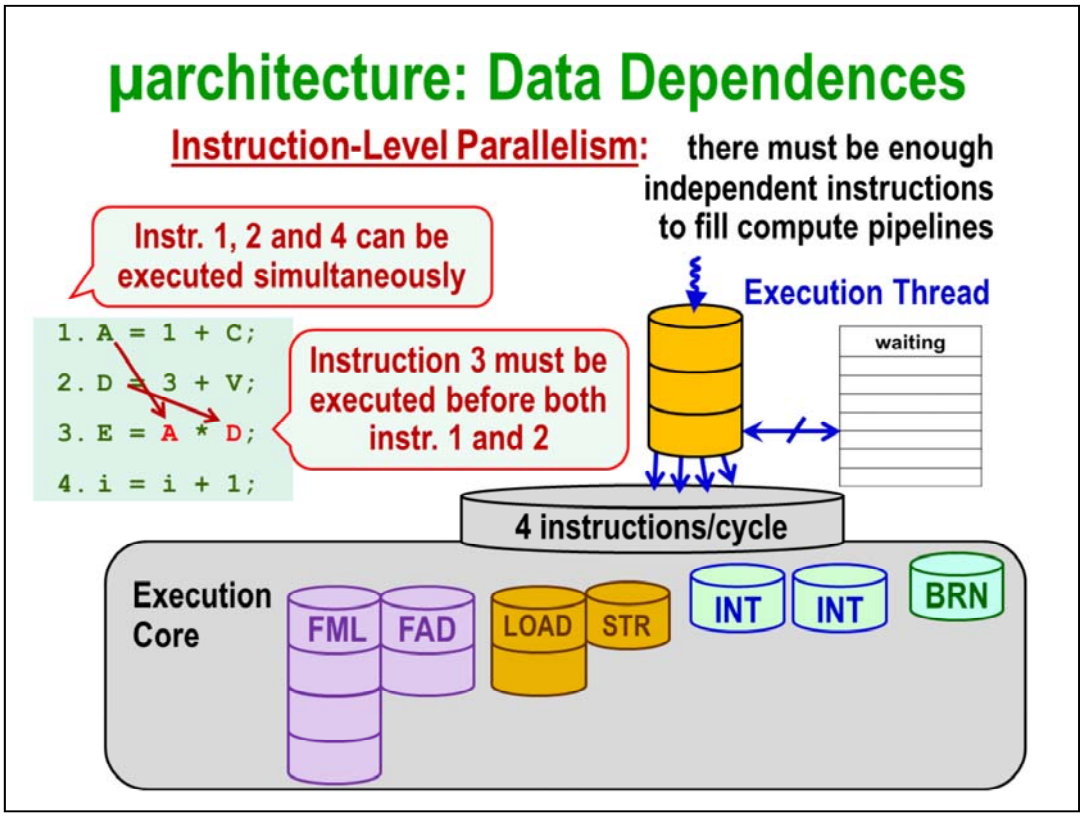
If 50% of the instructions in the inner loop of a program are of type FAD, since only one FAD instruction can start execution every cycle, then an IPC of at most 2 can be reached. We will say the execution performance is bounded to an IPC of 2 due to computation resource limitations: the program is compute-bounded or, more specifically, bounded by the floating-point addition capability of the processor.



The peak IPC of the processor is determined by the main entrance into the processor. However, the computational performance bottleneck of the system for a certain program may be one of the specific pipelines. The slide shows a table indicating the amount of instructions that must be executed for the version of the code shown in some previous slide, classified by instruction type. Working at the maximum rate of 4 instructions per cycle, the 16 instructions of each program's loop iteration could be executed in an average time of 16 / 4 = 4 clock cycles. However, since 5 of these instructions are FAD operations, and the maximum rate of FAD operations is one FAD per clock cycle (look at the table of processor throughputs), then each loop iteration should take at least 5 clock cycles. In this case, the pipeline of FAD operations would be the computational performance bottleneck of the application. The same argument can be used to say that the pipeline for executing LOAD operations needs 5 cycles to execute one loop iteration, and then is also a computational performance bottleneck.

Executing 16 instructions every 5 clock cycles means reaching an IPC of 16 / 5 = 3.2. If the actual execution exhibits an IPC of 0.9 then we can conclude that computation is not the performance bottleneck of the application, and then, there must be another bottleneck: which in most of the cases will be the memory system.

Detail: the CPU converts some instructions into one or more micro-instructions, and the table on the left indicates the number of micro-instructions (or sometimes known as micro-operations) classified by type. The actual program has 13 instructions (and 16 micro-operations), and the maximum IPC determined by limitations on the computation resources is 13 / 5 = 2.6, still far from an IPC of 0.9.



Another source of problems (also called hazards) that generates bubbles in the pipeline (and reduces performance) is the existence of data dependences among instructions. In the example above instruction 3 uses the results of instructions 1 and 2 (the outputs of instructions 1 and 2 are written to variables A and D, while they are also the inputs of instruction 3). It is not possible to execute the three instructions at the same time. On the contrary, instruction 4 is independent with respect to the other 3 instructions and could be executed simultaneously or even before instructions 1 to 3.

A processor that can execute instruction 4 before executing instruction 3 uses a technique that is referred to as "*out-of-order execution*" or "*dynamically scheduled execution*". Most CPUs implement the ability of "out-of-order execution". This characteristic does not come for free: there is a lot of hardware control required, and also a substantial amount of additional energy consumption. For this reason, GPUs and other low-energy consumption processors are not able to dynamically reorder the execution of instructions. In this latter case, the execution of instruction 4 must wait until instruction 3 can be executed.

As a consequence of how processors execute instructions, the order of instructions in the code is very important and may determine a good or a low processor throughput (or IPC)

Example: Minimum Execution Time

```
float VectorAdd ( float *A, const int N) {
  int i; float S= 0.0;
  for (i=0; i<N; i++)
    S = S + A[i];
  return S;
}
```

Level:	Add Operation
Latency:	4 clock cycles
Throughput:	1 per clock cycle

Additions are performed sequentially

Data Dependence Path:

((...(((S + A[0]) + A[1]) + A[2]) + A[3]) + ...)+ A[N-1])

Time of Critical Path

Latency(ADD) + Latency(ADD) + ... ≈ N×Latency(ADD) = 4N cycles

This code specifies a completely sequential execution of the floating-point addition operations, as specified by the parenthesis in the expression bellow. As a consequence of the sequential execution, the minimum execution time of the program will be the latency of a floating-point addition multiplied by the number of elements in the vector.

If N tasks have to the executed sequentially, then you need a minimum time which is the sum of the latencies to perform all the tasks.

Since the latency of each floating-point addition is 4 cycles, executing one instruction after one another means executing one floating-point addition every four clock cycles, which is 25% of the maximum throughput of the computer. Therefore, with this level of parallelism (1) we do not saturate the FAD computation resource and then we say that performance is limited by data dependences.

Example 2: Minimum Execution Time

```
float VectorAdd ( float *A, const int N) {
  int i; float S1,S2,S3,S4; S1=S2=S3=S4= 0.0;
  for (i=0; i<N; i+=4) {
    S1 += A[i]; S2 += A[i+1]; S3 += A[i+2]; S4 += A[i+3]);
  }
  return S1+S2+S3+S4;
}
```

Level:	Add Operation
Latency:	4 clock cycles
Throughput:	1 per clock cycle

4 Additions are performed sequentially

Data Dependence Path:

((...((S1 + A[0]) + A[4]) +A[8] ...)) ((...((S2 + A[1]) + A[5]) + A[9]...))
((...((S3 + A[2]) + A[6]) +A[10] ...)) ((...((S4 + A[3]) + A[7]) + A[11]...))

Time of Critical Path

≈ N/4×Latency(ADD) = N cycles

This "optimized" code specifies an execution of the floating-point addition operations that provides more parallelism (more ILP), as specified by the parenthesis in the expressions bellow. An average of up to 4 floating-point additions can be executed simultaneously (or can overlap their execution). Therefore, the minimum execution time of the program will be the latency of a floating-point addition multiplied by the number of elements in the vector divided by four.

Since the latency of each floating-point addition is 4 cycles, executing 4 instructions simultaneously means executing one floating-point addition per clock cycle, which is exactly the maximum throughput of the computer. Therefore, with this level of parallelism (4) we can saturate the computation resource and then make performance compute-bound.

Example 2B: Minimum Execution Time

```
float VectorAdd ( float *A, const int N) {
  int i; float S= 0.0;
  for (i=0; i<N; i+=4)
    S = S + (A[i] + A[i+1] + A[i+2] + A[i+3]);
  return S;
}
```

4 Additions are performed sequentially

Level:	Add Operation
Latency:	4 clock cycles
Throughput:	1 per clock cycle

Data Dependence Path:

((...((S + (A[0] + A[1] + A[2] + A[3]) + (A[4]+A[5]+A[6]+A[7]) ...))

Time of Critical Path

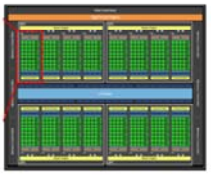
≈ N/4×Latency(ADD) = N cycles

This version of the code also provides parallelism for executing up to 4 floating-point addition operations simultaneously. It is not easy to see that, but a processor that can reorder the execution of instructions will be able to execute at this rate.

One important question: *Is the compiler able to do this kind of optimization?* The answer is that it is not safe to do that, because changing the order of the additions may change the results (when using floating-point representation). So, in order to allow the compiler perform this kind of optimization you must use the **`-Ofast`** flag (where you give the compiler permission to generate code that does not exactly generates the same result).

In some cases, the compiler is not "smart" enough to make this kind of optimization when it would be a good performance improvement.

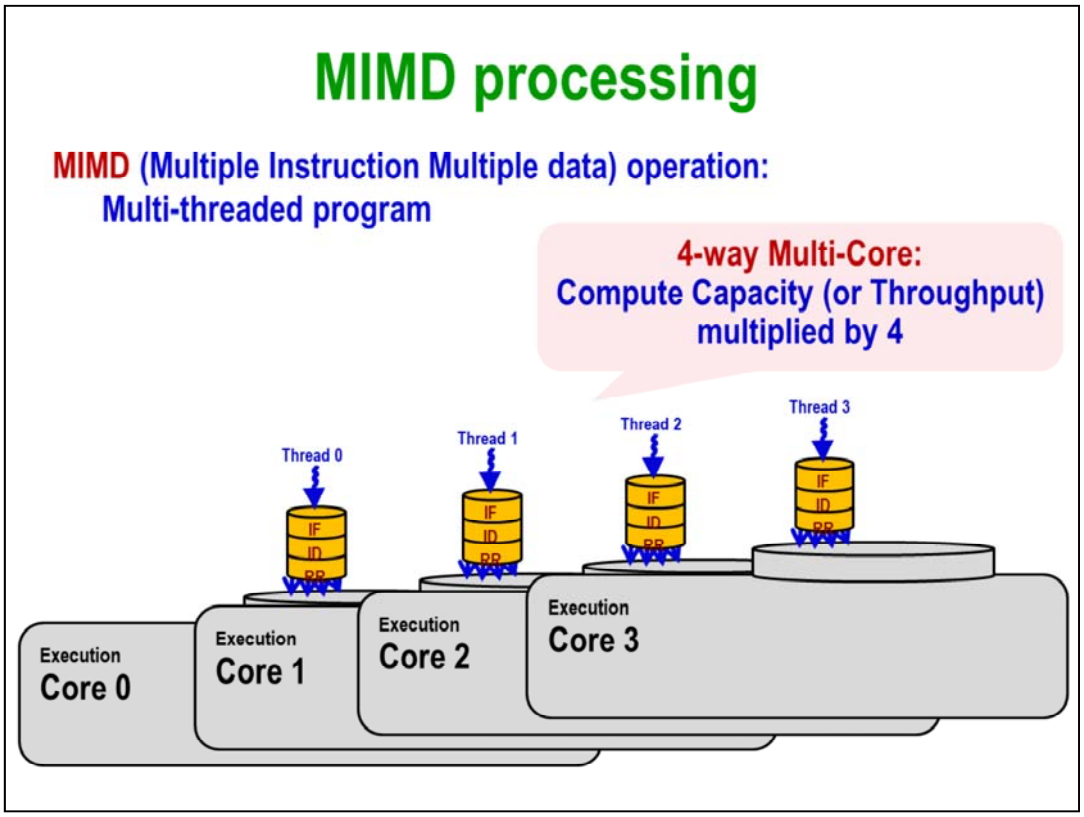
Processor Architecture



Processor Details to understand Performance

MIMD (Multiple Instruction Multiple data) operation

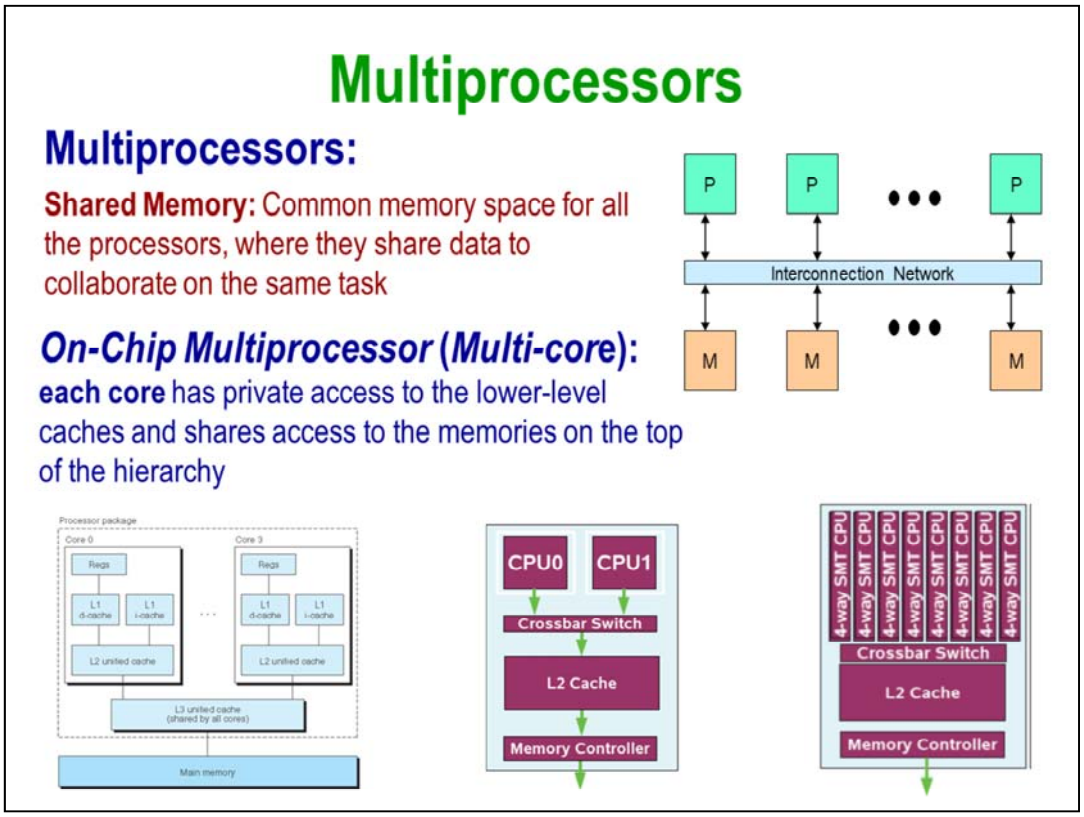
Finally, we will describe a final strategy to use parallelism in order to improve performance: using multiple execution cores simultaneously, which is an architecture paradigm called MIMD (Multiple-Instruction Multiple Data) processing.



A multicore processor is a processor die containing several compute elements that work independently, called computation cores, each one executing a single program or thread of instructions. The multicore processor can only be exploited efficiently if it runs as many independent threads as computation cores. The creation of independent threads is mostly the responsibility of the programmer (currently, compilers offer little help on automatic thread parallelization, mostly on very simple code fragments).

Do not confuse the computation cores described here and depicted in this slide with the execution pipelines (or sometimes called functional units) depicted in the previous slides. Each computation core contains several execution pipelines, and each processor contains multiple computation cores.

Duplicating computation capacity (or throughput) is the easy part; the complex part is how to duplicate memory bandwidth, without increasing memory access latency significantly.

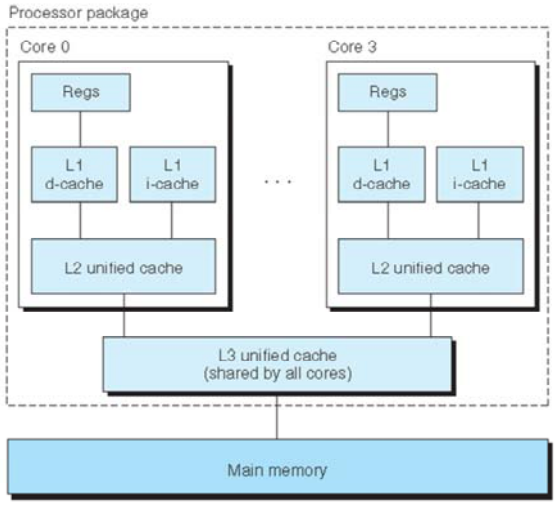


A multiprocessor is a system with several computation elements that share the same memory address space: one computation core can write a value to some memory location and a different computation core can later read that location and get the value. Such a system can be composed of several processor dies, each containing several computation cores. A single application can solve a problem by creating several independent threads of execution and distributing the total amount of work among the threads. These threads can cooperate using the shared memory space by writing and reading data to communicate information and to synchronize their activities.

A common strategy to assure that the peak memory bandwidth available for each computation core remains the same regardless of the total number of computation cores in the system is to use private memory caches; i.e., each computation core has its own private cache (or hierarchy of caches) and most of the time retrieves the required data from its private cache memories.

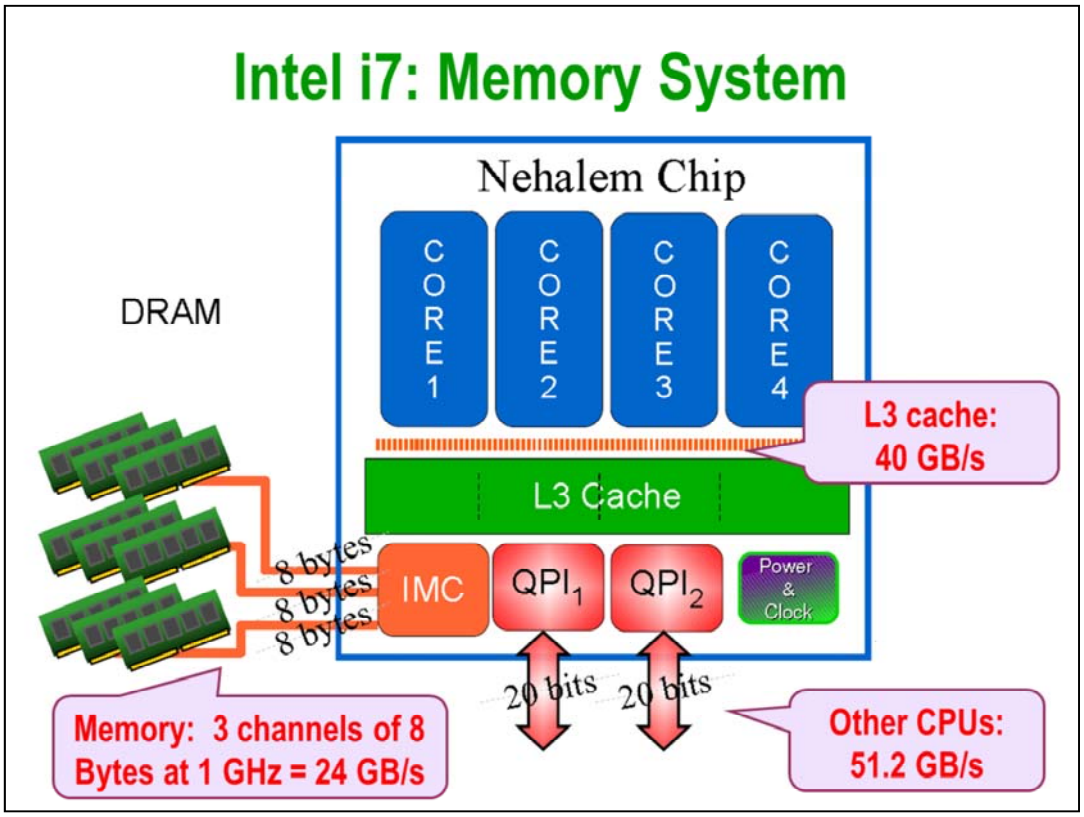
However, many applications process large data sets, and the work assigned to each thread is so large that private local cache memories cannot hold most of the data required locally. For those applications, it is necessary to support a high bandwidth DRAM memory system. In a few slides we will describe some approaches for supporting a scalable shared memory system.

Memory Hierarchy: intel i7 950

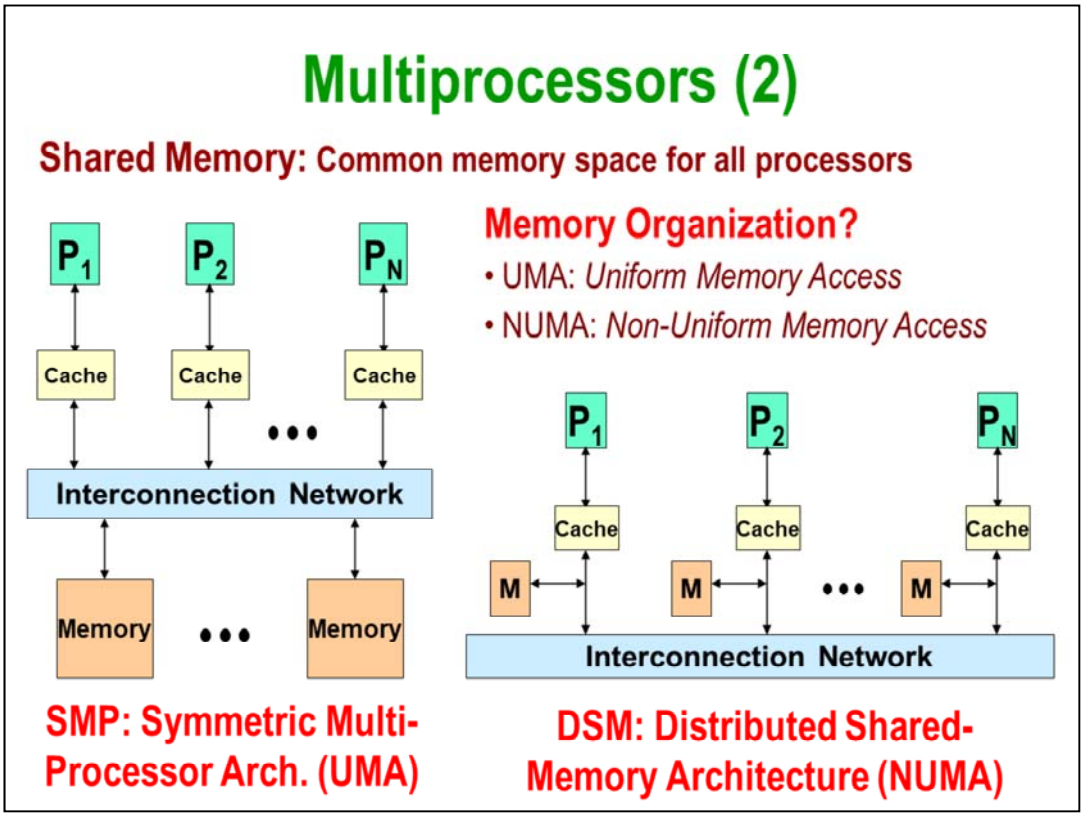


Cache type	Access time (cycles)	Cache size (C)
L1 i-cache	4	32 KB
L1 d-cache	4	32 KB
L2 unified cache	11	256 KB
L3 unified cache	30–40	8 MB

Here we have a description of the memory hierarchy of the i7 950 processor. It has three levels of memory: private L1 and L2 caches for each computation core, and a larger, L3 cache and main memory which are shared by all the computation cores.



The read and write bandwidth to the L3 cache is 40 GB/sec, while the peak bandwidth to the main memory is 24 GB/sec. Several processor chips can be connected together to build a more powerful multiprocessor. Each link used for processor interconnect has a peak bandwidth of 51.2 GB/sec. The performance of an application can be sometimes constrained by one of these bandwidth limits.



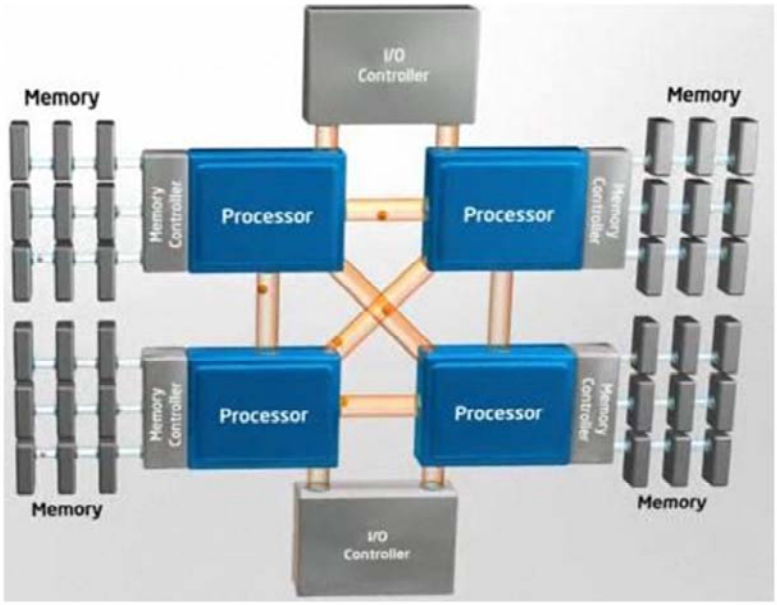
For applications that process large data sets, the work assigned to each thread is often too large to fit in the private local cache memories, and it is necessary to support a high bandwidth DRAM memory system.

When increasing the number of processors, P, we must maintain the average DRAM bandwidth per processor, or otherwise the ratio between computation throughput and memory bandwidth will degrade: computation elements will be more and more idle, waiting for the data.

A UMA system (Symmetric Multiprocessor Architecture) maintains bandwidth per computation element by using a more and more complex interconnection network, which increases cost more than linearly (typically between $P \times \log P$ and P^2), while the latency also increases (typically with $P \times \log P$).

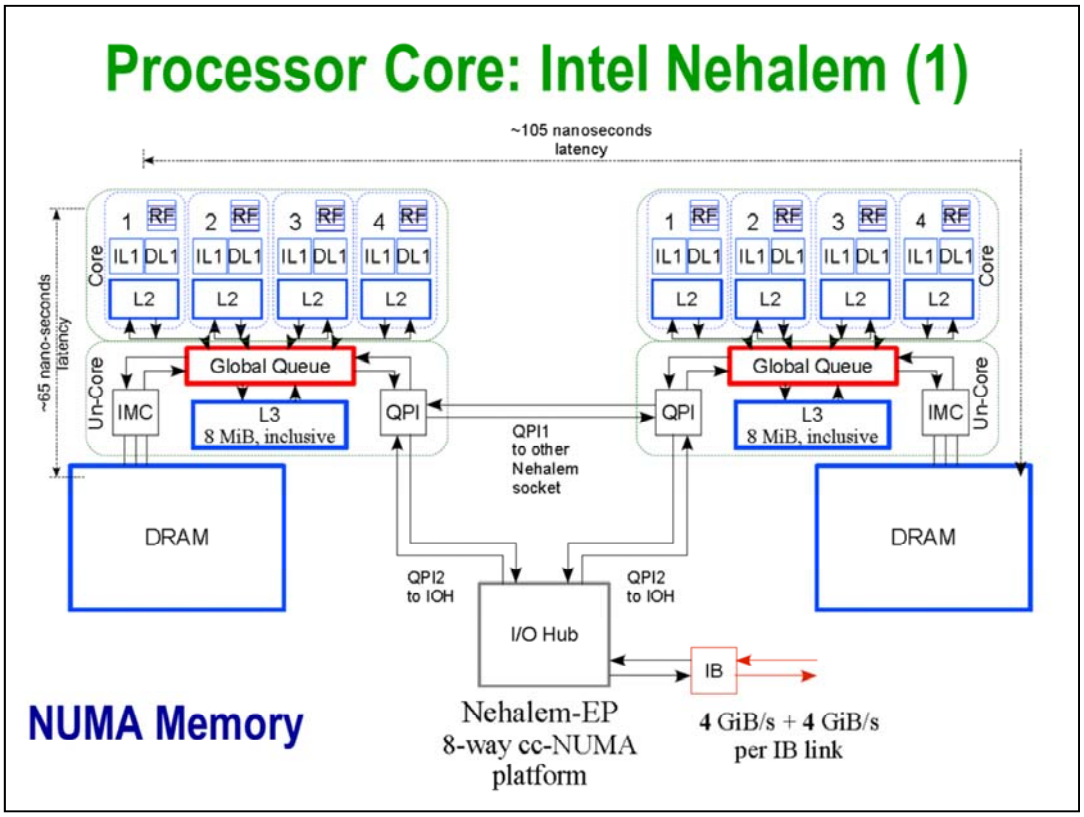
Instead, a NUMA system (Distributed Shared-Memory Architecture) maintains the bandwidth (and achieves low latency) to the local memory (a piece of the whole DRAM memory in the system that is directly attached to each processor element), but bandwidth to remote memory (the remaining DRAM memory, accessed through a complex interconnection network) is decreased (and latency is increased). The responsibility of using a NUMA system efficiently shifts to the programmer, who must place the data near to the processor that consumes those data. The efficient data management may involve duplication of information and migration of data.

Intel: *QuickPath point-to-point connection*



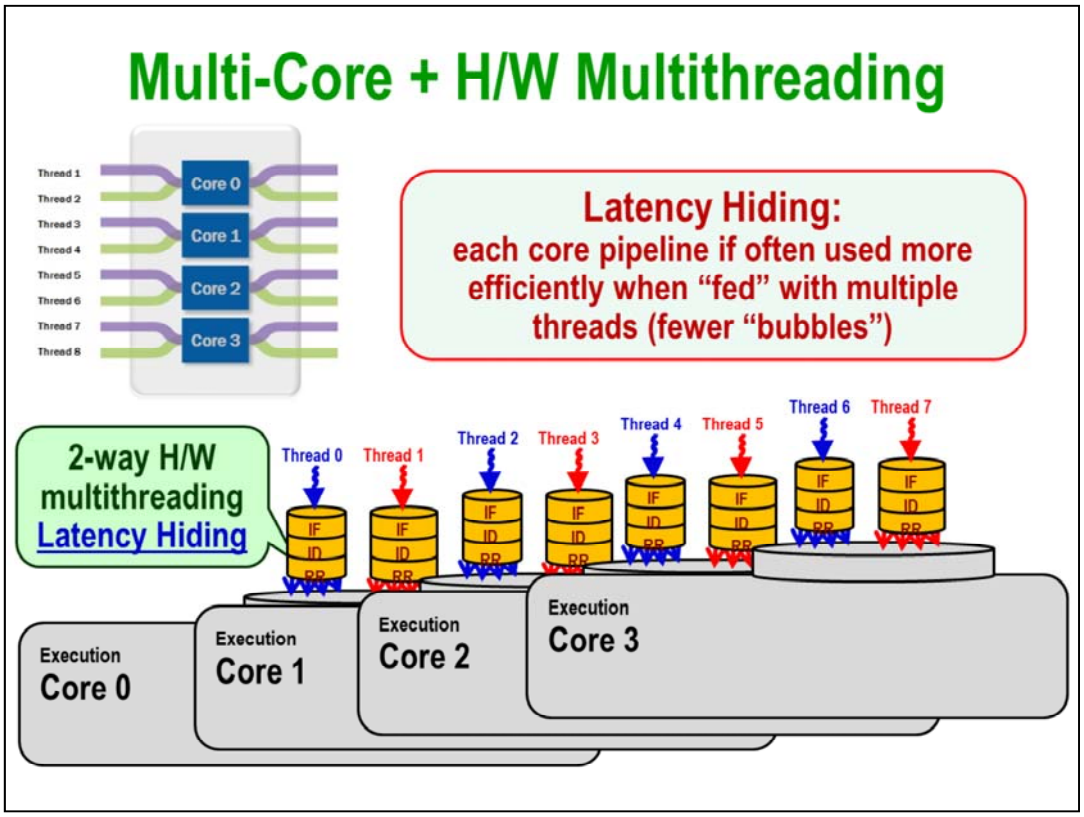
NUMA

This is an example of a NUMA architecture build using four intel Nehalem processors. Accessing the local memory requires 0 communication hops, but accessing any remote memory requires one communication hop through a direct connection between processors using a QuickPath link.



This Figure provides more detail of the elements composing a NUMA architecture. A global queue holds memory requests that cannot be served by the private cache memories of a computation core. Those requests check in the L3 cache tags where the data can be found: (1) in the local L3 cache; (2) in the local DRAM; or (3) in a remote processor. In the last case, a request message is sent along the Quick Path Interface (QPI): the data can be found in the L3 cache or the DRAM of the remote processor.

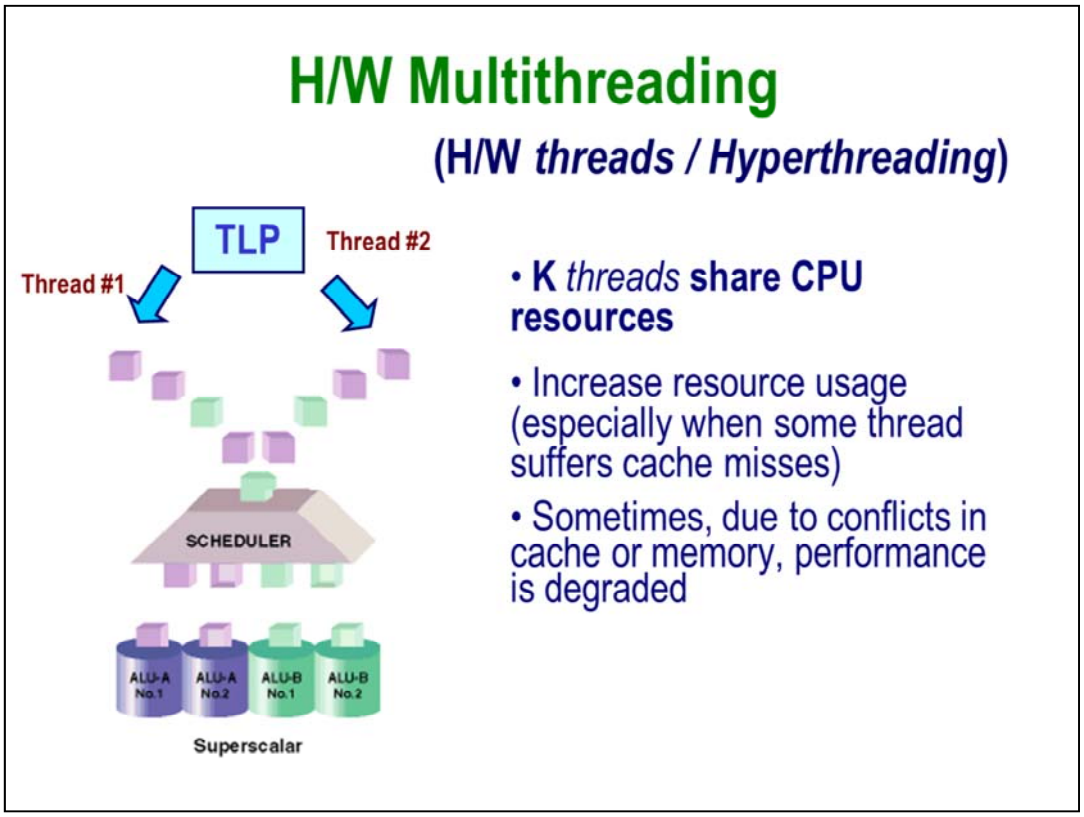
In the Figure we can see that there is an extra delay of 40 ns when the data must be retrieved from a remote DRAM bank rather than a local DRAM bank.



Some multicore processors include hardware support for executing two (or more) threads simultaneously within the same computation core. This capability is called H/W multithreading (or *hyperthreading*). The threads that are executed in the same computation core share the execution pipelines and other resources.

The intention of using two or more threads per computation core is to increase the available parallelism: the computation core now have more opportunities to find an independent instruction (not waiting for previous results) and then maximize the utilization of the execution pipelines. A good scenario occurs when the threads executing in the same core are *complementary*, for example one of them saturating the memory execution pipeline and the other one saturating the floating-point computation pipelines.

It is not always necessary to execute two or more threads in the same core to achieve maximum performance, and in some situations it can even be counterproductive, since the threads compete for the space in the cache memories that are private to the computation core and may generate pollution that ends degrading the performance of the execution of both threads.

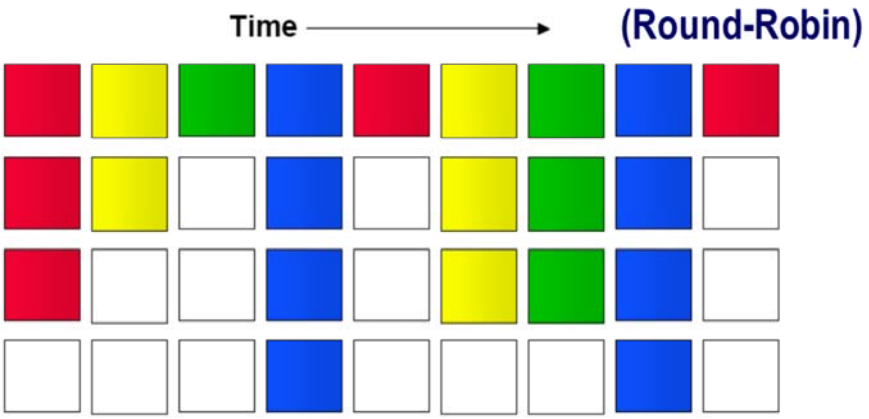


This is an example scheme showing how the execution pipelines are shared by the instructions flowing from both threads of execution. The instruction execution scheduler has more opportunities to fill the execution pipes when two or more independent threads are available. But remember that H/W multi-threading is only effective when the threads sharing the computation core are complementary and are not competing for the same computation or memory resource.

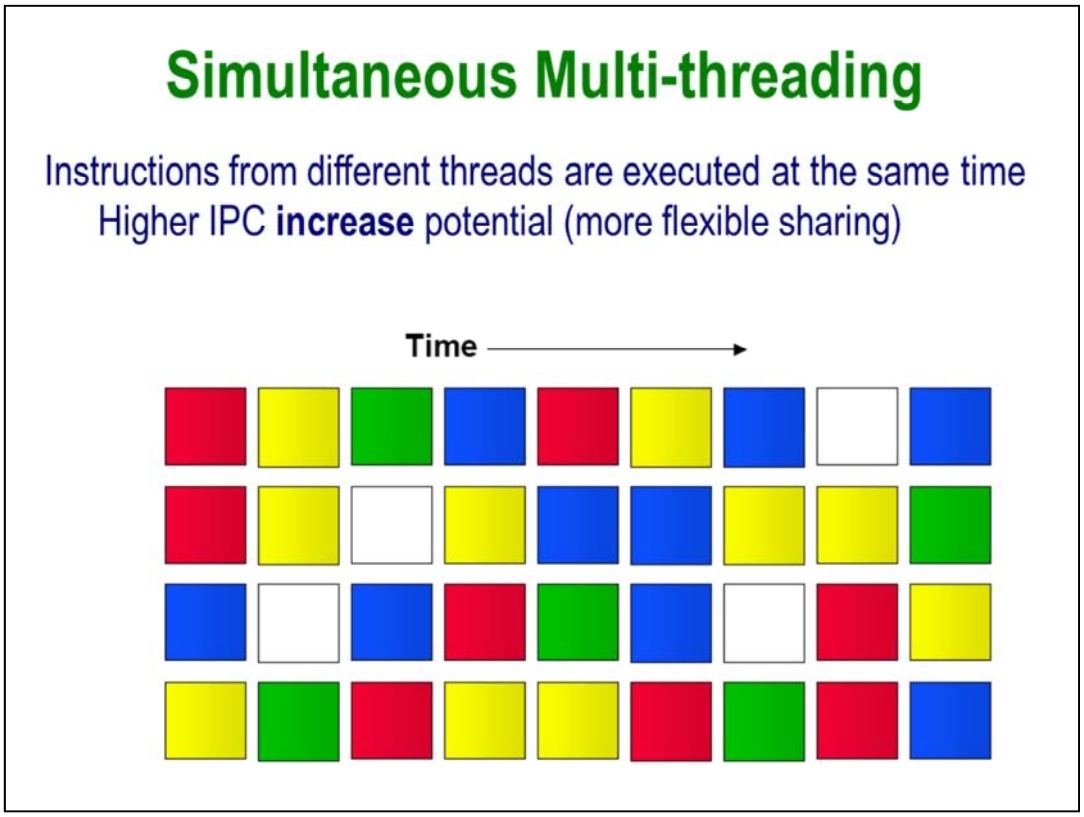
The amount of threads that may be running simultaneously in a given processor often defines the so-called amount of logical CPUs in the system, while the number of computation cores that can operate simultaneously (and independently) defines the amount of physical CPUs in the system.

Fine-Grained Multi-threading

Each cycle a different thread can execute instructions
(each thread has a different color)
IPC **increases** by hiding some latencies



The sharing of resources among the threads can be cycle by cycle, so that every clock cycle only instructions of a single thread (color) can enter the execution pipelines (while instructions of different threads may also be still executing inside those pipelines) or …



… the sharing of resources among the threads can be simultaneous, so that every clock cycle instructions of different threads (colors) can enter the execution pipelines. This last scheme provides maximum flexibility and is the strategy used by the intel Nehalem computation core that is present in the processors of our laboratory.