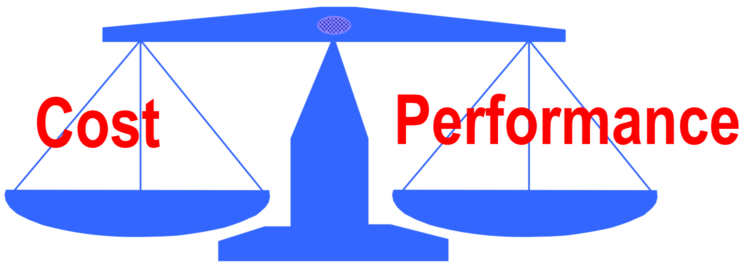
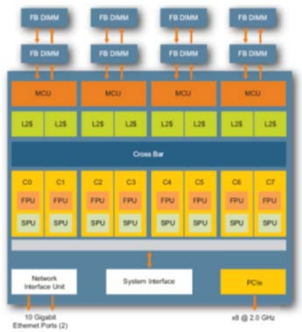
Parallel Programming





Performance Engineering

Real application case: Laplace2D with Jacobi Method

juancarlos.moure@uab.es Office: QC-3034

Laplace Equation: Problem Description

2D Laplace Equation

$$\vec{\nabla}^2\Phi(x,y) = \frac{\partial^2\Phi(x,y)}{\partial x} + \frac{\partial^2\Phi(x,y)}{\partial y} = 0$$

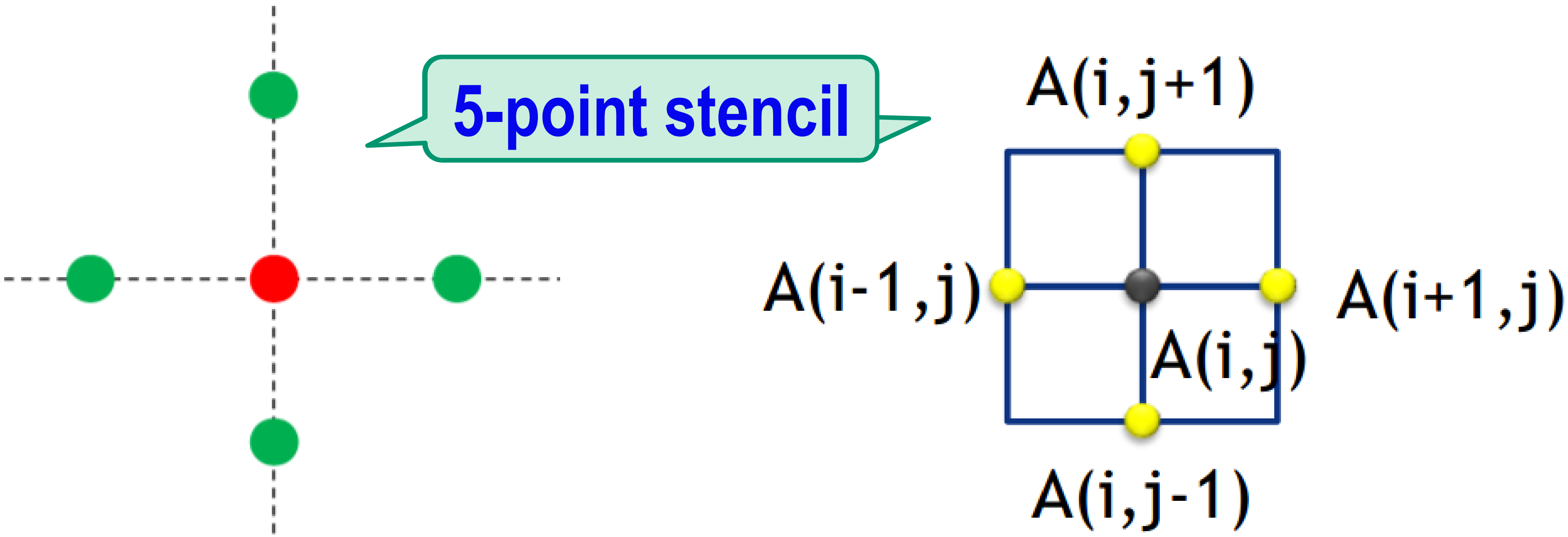
Discretized Approximation

$$\left(\frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{h^2}\right) + \left(\frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{h^2}\right) \approx 0$$

$$\Phi_{i,j} \approx \frac{1}{4}\left[\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}\right]$$

Jacobi Iteration: Problem Description

Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Computational Solution (1)

```
#define n 4096
#define m 4096

float A[n][m], Anew[n][m], tol, error;
int i, j, iter_max, iter=0;
…
while ( error > tol && iter < iter_max )
{
  for( i=1; i < m-1; i++)
    for( j=1; j < n-1; j++ )
      Anew[j][i]= (A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])/4;


  error = 0.0f;
  for( i=1; i < m-1; i++)
    for( j=1; j < n-1; j++ )
      error = fmaxf( error, sqrtf(fabsf(Anew[j][i]-A[j][i])));

   …
}
```

Iterate until error is small enough or too much iterations

Use state of A before updating A

Error is maximum square root of difference

Computational Solution (2)

```
…
while ( error > tol && iter < iter_max ) {
 for( i=1; i < m-1; i++)
    for( j=1; j < n-1; j++ )
       Anew[j][i]= (A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])/4;


  error = 0.0f;
  for( i=1; i < m-1; i++)
    for( j=1; j < n-1; j++ )
      error = fmaxf( error, sqrtf(fabsf(Anew[j][i]-A[j][i])));

  for( i=1; i < m-1; i++ )
    for( j=1; j < n-1; j++)
       A[j][i] = Anew[j][i];

  iter++;
  if (iter % (iter_max/10) == 0)
      printf("%5d, %0.6f\n", iter, error);
}
```

Update state of A using state in Anew (temporal matrix)

Print error during execution

Optimizations

Apply optimizations in this order

1) Loop fusion:

```
compute new cell and error in the same loop
```

2) Loop interchange

```
for( j=1; j < n-1; j++ )
  for( i=1; i < m-1; i++ ) {
    Anew[j][i]= (A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])/4; … }
for( i=1; i < m-1; i++ )
  for( j=1; j < n-1; j++ )
     A[j][i] = Anew[j][i];
```

3) Strength Reduction:

```
    Anew[j][i]= (A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])*0.25f;
```

4) Code Motion: remove sqrtf() from inner loop and move outside

```
    error = fmaxf( error, fabsf( Anew[j][i]-A[j][i] ) ) )
```

5) Double Buffer: avoid copying from Anew to A by reversing roles of A and Anew on every iteration

GOAL: measure performance metrics (with iter_max=100), identify relevant performance facts, and explain results (as much as possible)

Additional Work

6) **Modify size of matrix and total iterations and check performance**

```
gcc -Ofast -lm -Dm=1024 -Dn=512 laplace2d.c …
icc –O3 -Dm=1024 -Dn=512 laplace2d.c …
```

Use your best performing code

Problem Size (n,m,iter)	wall-clock time
(4096, 4096, 100)	
(4096, 4096, 1000)	
(512, 1024, 100)	
(512, 1024, 1000)	
(512, 1024, 10000)	
(512, 1024, 100000)	

GOAL: measure effect on performance of problem size & explain results (as much as possible)

Compiling with GCC and ICC

Module: install directory paths for executables, libraries and include files in the LAB computers with Linux OS

```
$ module add gcc/6.1.0
$ gcc -Ofast -lm laplace2d.c -o L2Dg
```

Make best optimizing effort

```
> module add intel/16.0.0
> icc -Ofast laplace2d.c -o L2Di
```

Executing & Measuring Performance

Command to instrument execution

Performance metrics to measure

iter_max declared at run-time

```
$ perf stat -e cycles,instructions,cache-misses,task-clock ./L2Dg 100
Jacobi relaxation Calculation: 4096 x 4096 mesh, maximum of 100 iterations

   10, 0.155006
   20, 0.110024
   30, 0.089901
   40, 0.077374
   50, 0.069623
   60, 0.063285
   70, 0.058825
   80, 0.054945
   90, 0.051831
  100, 0.049208


 Performance counter stats for './L2Dg 100':

   289.757.998.767      cycles                   #    3,367 GHz
    28.377.296.648      instructions             #    0,10  insns per cycle
     5.631.065.118      cache-misses             #   65,433 M/sec
      86057,833003      task-clock (msec)        #    1,001 CPUs utilized

      85,940821112 seconds time elapsed
```

Mesh size is declared at compile-time

Error after 100 iterations

Performance metrics of execution

Processor Performance

$$\boxed{\text{CPU Time} = \text{clock cycles} \times \text{clock cycle time}}$$

$$\boxed{\text{CPU Time} = \text{instructions} / (\text{IPC} \times \text{clock frequency})}$$

clock cycle time = 1 / clock frequency

IPC: ratio of Instructions executed Per clock Cycle

cache miss: the data requested by the program do not reside in the internal memory of the processor chip and must be retrieved from main memory (DRAM). Cache misses degrade the IPC metric

Final Optimized Code

Code Motion

Double Buffer

Loop Interchange

Strength Reduction

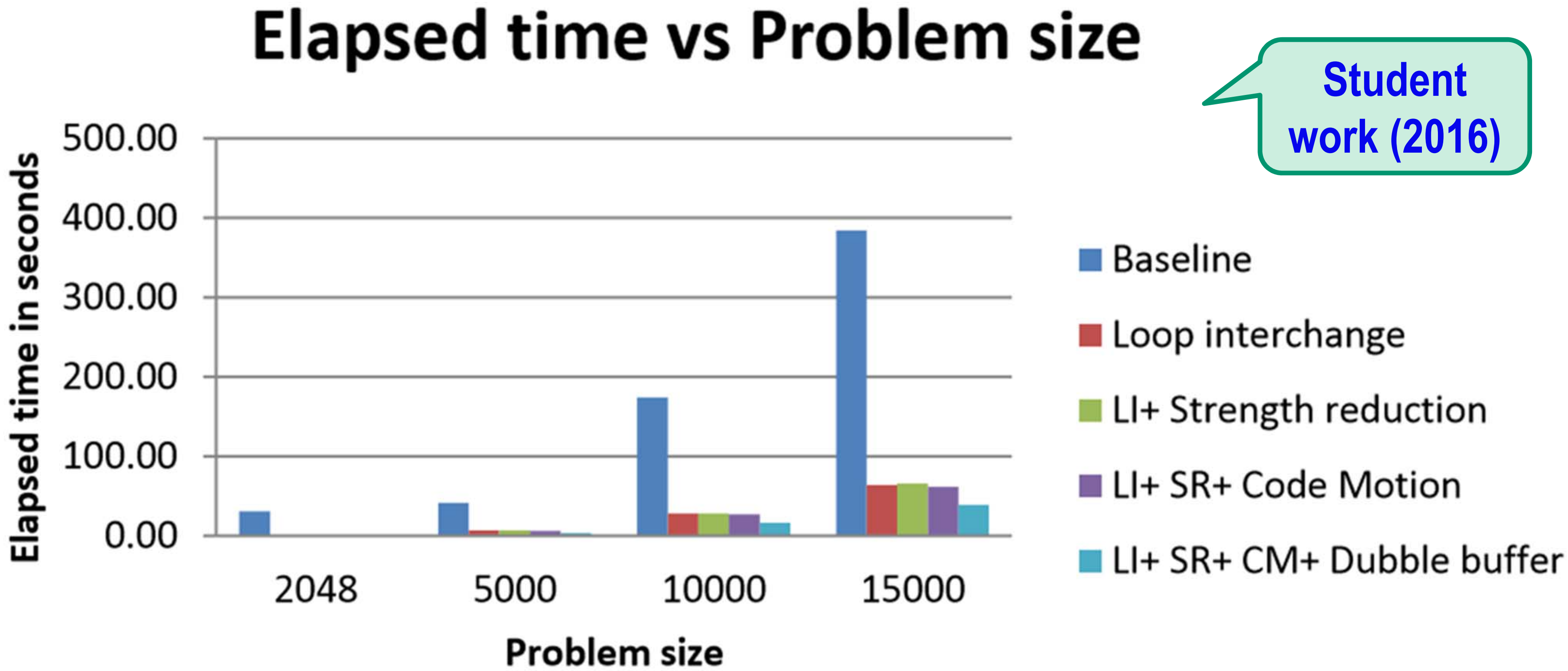
Loop Fusion

Double Buffer

Code Motion

```
while ( err > tol*tol && iter < iter_max ) {
  err = 0.0f;
  if (iter%2 == 0)
   for( j=1; j < n-1; j++ )
     for ( i=1; i < m-1; i++) {
       Anew[j][i]= (A[j][i+1]+A[j][i-1]+
                    A[j-1][i]+A[j+1][i])*0.25f;
       err = fmaxf( err, fabsf(Anew[j][i]-A[j][i]));
     }
  else
   for( j=1; j < n-1; j++ )
     for ( i=1; i < m-1; i++) {
       A[j][i]= (Anew[j][i+1]+Anew[j][i-1]+
                 Anew[j-1][i]+Anew[j+1][i])*0.25f;
       err = fmaxf( err, fabsf(Anew[j][i]-A[j][i]));
     }
  if (++iter % (iter_max/10) == 0)
     printf("%5d, %0.6f\n", iter, sqrtf(err));
}
```

How to visualize (analyze) Performance



Student work (2016)

Elapsed time is not a good metric to compare variations on problem size

$$\text{Performance} = \frac{\text{Total Work}}{\text{Second}}$$

Better Metric
Normalized by Work

Work per second: Higher is Better

$$\frac{\text{Work}}{\text{Second}} = \frac{\text{Operations}}{\text{Instruction}} \quad \frac{\text{Instructions}}{\text{Clock Cycle}} \quad \frac{\text{Clock Cycles}}{\text{Second}}$$

OpRate IPC Clock Freq.

Coding Efficiency: the more work of the program done by a fixed number of processor's machine instructions, the better

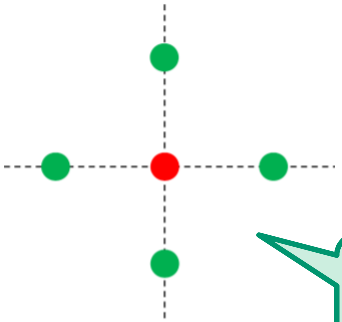
MicroArchitecture Throughput: The more machine instructions executed every clock cycle, the better

H/W speed: The higher the clock frequency of the processor, the better

All factors are better when they are higher
The influence of the problem size is avoided
Programmers can improve OpRate & IPC

Algorithmic Efficiency: Complexity

```
Anew[j][i]= (A[j][i+1]+A[j][i-1]+
             A[j-1][i]+A[j+1][i])*0.25f;
error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
```



5-point stencil

Constant WORK: on each iteration of inner loop

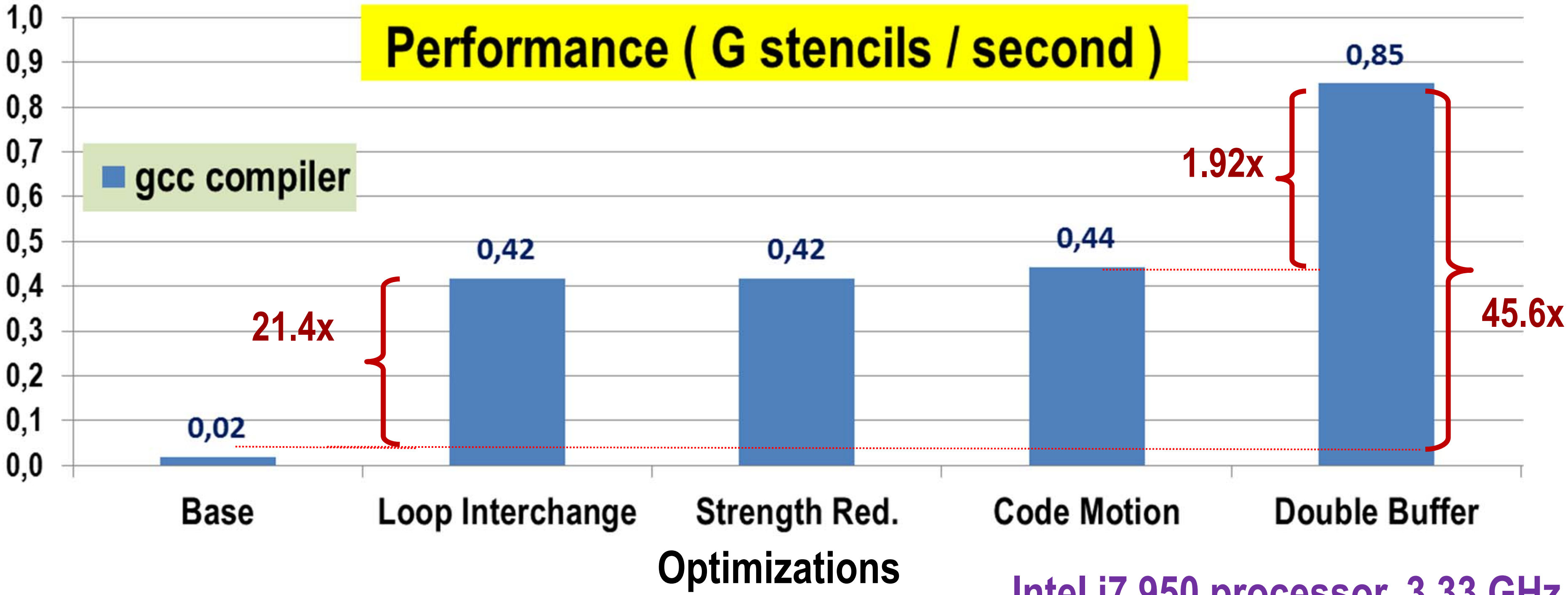
 # Stencil operations: $iter \times n \times m$

<u>Algorithmic Complexity or Total Work</u>:

$\theta\,(iter \times n \times m)$

Comparing Performance: Speedup

Problem Size = 2K × 2K matrix, 100 convergence iterations

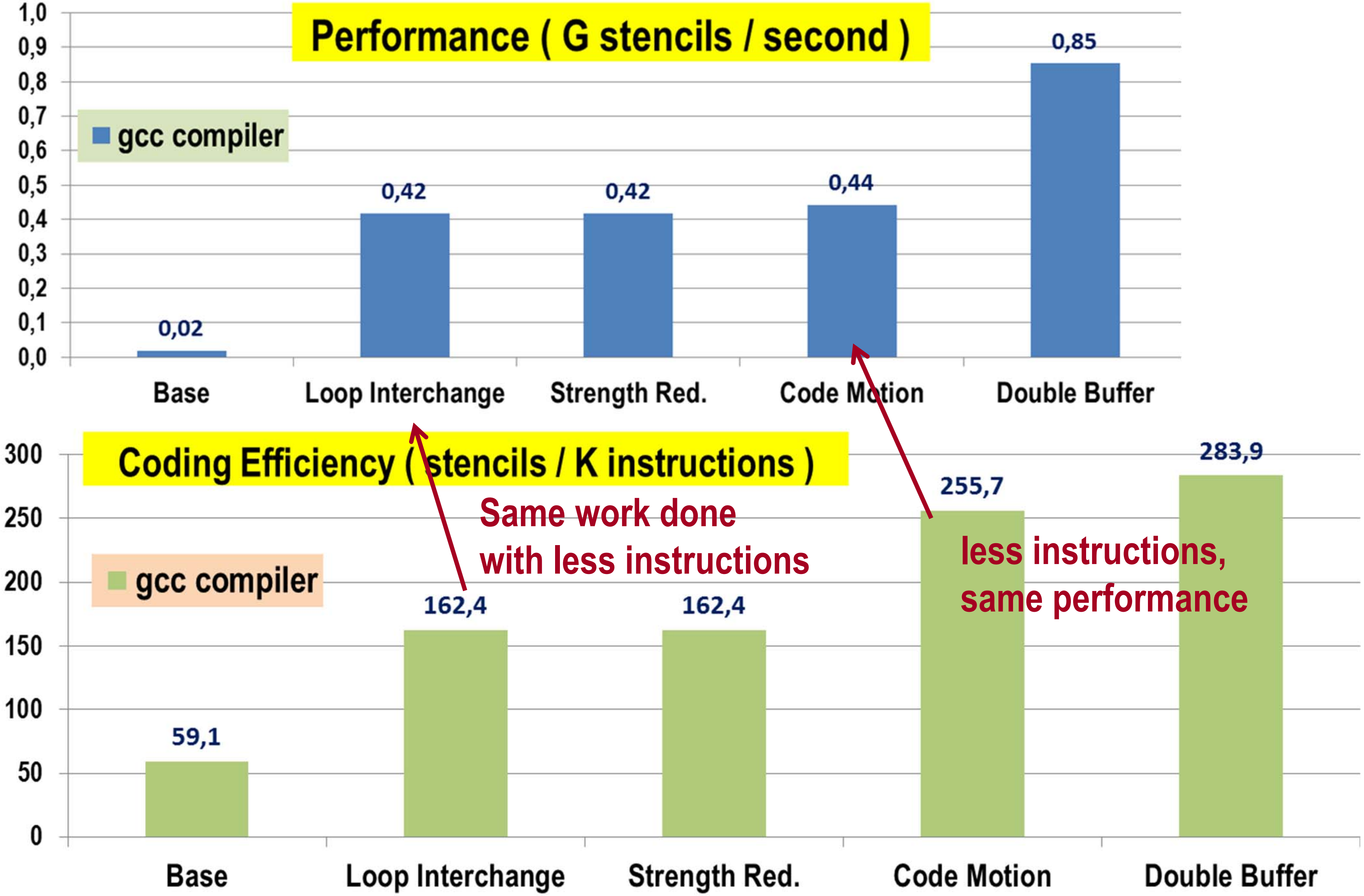


Intel i7 950 processor, 3.33 GHz

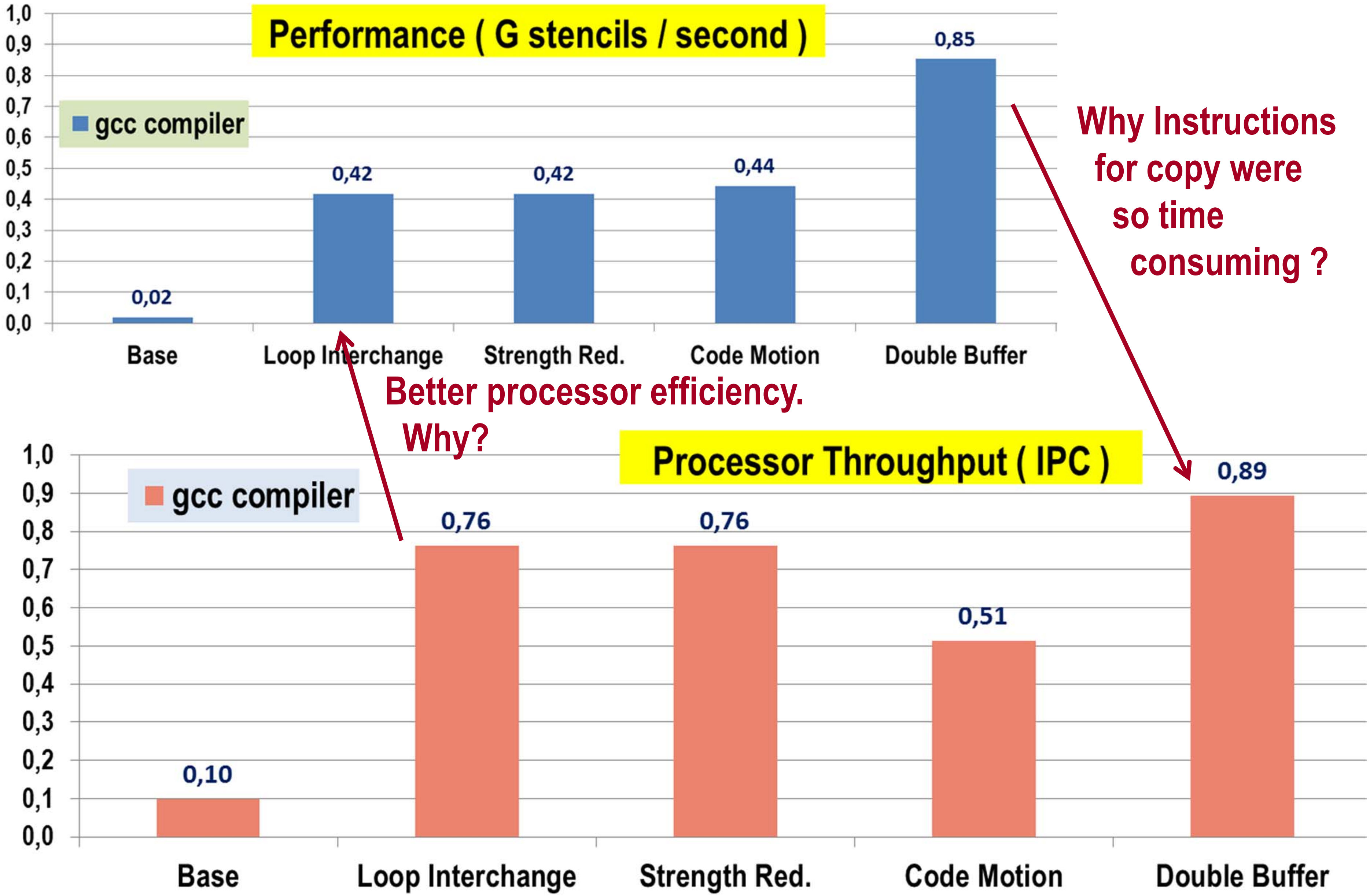
Loop interchange:	very large improvement (speedup is 21.4x)
Double Buffer:	almost doubles performance (1.92x)
Strength Red. & Code motion:	very small or no improvement

Speedup:
$Perf_{NEW} / Perf_{OLD}$

Explaining Speedup: Coding Efficiency



Explain: Processor Throughput



Profile & Assembly Inspection

```
$ module add gcc/6.1
$ gcc –Ofast -lm Laplace2D.c –o L2Dg
$ perf record ./L2Dg
$ perf report
```

generates file perf.data

uses file perf.data

```
 99,70%  saxpy    L2Dg                 [.] main
  0,10%  saxpy    [kernel.kallsyms]    [k] clear_page_c
  0,03%  saxpy    [kernel.kallsyms]    [k] apic_timer_interrupt
  0,02%  saxpy    [kernel.kallsyms]    [k] task_tick_fair
  0,01%  …
```

Press enter here to visualize annotated code

Assembly Code: analysis & comparison

```
 0,23 │1c8:  movss   (%rsi),%xmm4
 1,17 │      add     $0x2,%r11d
      │      addss   (%rcx),%xmm4
 1,22 │      movss   (%rdx),%xmm5
42,47 │      addss   %xmm5,%xmm0
 1,31 │      movss   0x4000(%rdx),%xmm7
22,47 │      add     $0x8000,%rsi
 0,03 │      add     $0x8000,%rcx
 0,14 │      add     $0x8000,%rdx
 0,03 │      add     $0x8000,%rax
 0,26 │      addss   %xmm0,%xmm4
 0,17 │      movss   -0x4000(%rcx),%xmm0
 0,94 │      addss   -0x4000(%rsi),%xmm0
 1,08 │      mulss   %xmm3,%xmm4
 0,43 │      movss   %xmm4,-0x8000(%rax)
 0,09 │      subss   %xmm6,%xmm4
 0,06 │      addss   %xmm7,%xmm6
 0,60 │      andps   %xmm2,%xmm4
 0,06 │      addss   %xmm6,%xmm0
 0,43 │      movaps  %xmm7,%xmm6
      │      sqrtss  %xmm4,%xmm4
 0,66 │      mulss   %xmm3,%xmm0
 0,97 │      movss   %xmm0,-0x4000(%rax)
 0,17 │      subss   %xmm5,%xmm0
 0,51 │      cmp     $0xffd,%r11d
 0,03 │      andps   %xmm2,%xmm0
 0,34 │      sqrtss  %xmm0,%xmm0
 1,99 │      maxss   %xmm4,%xmm0
 0,60 │      maxss   %xmm0,%xmm1
 0,74 │      movaps  %xmm5,%xmm0
      │      jne     1c8
```

Loop Fusion

gcc –Ofast (v 6.1)

```
 7,04 │290:  movups  (%r9,%rax,1),%xmm0
 0,19 │      movaps  %xmm6,%xmm10
      │      movups  (%rdx,%rax,1),%xmm9
 1,97 │      addps   %xmm9,%xmm0
 5,82 │      addps   (%r8,%rax,1),%xmm0
 0,94 │      addps   (%rdi,%rax,1),%xmm0
11,73 │      mulps   %xmm5,%xmm0
 0,47 │      movaps  %xmm0,(%rcx,%rax,1)
 0,28 │      subps   (%rsi,%rax,1),%xmm0
 6,85 │      add     $0x10,%rax
 0,09 │      cmp     $0x3fe0,%rax
      │      andps   %xmm4,%xmm0
 6,19 │      cmpneq  %xmm0,%xmm10
 0,75 │      rsqrtp  %xmm0,%xmm9
 1,03 │      andps   %xmm10,%xmm9
 0,56 │      mulps   %xmm9,%xmm0
 7,97 │      mulps   %xmm0,%xmm9
 5,16 │      mulps   %xmm2,%xmm0
 2,44 │      addps   %xmm3,%xmm9
 6,10 │      mulps   %xmm9,%xmm0
20,26 │      maxps   %xmm0,%xmm1
13,23 │      jne     290
```

Loop Interchange

mulps: Multiply Packed Single-Precision Floating-Point Values

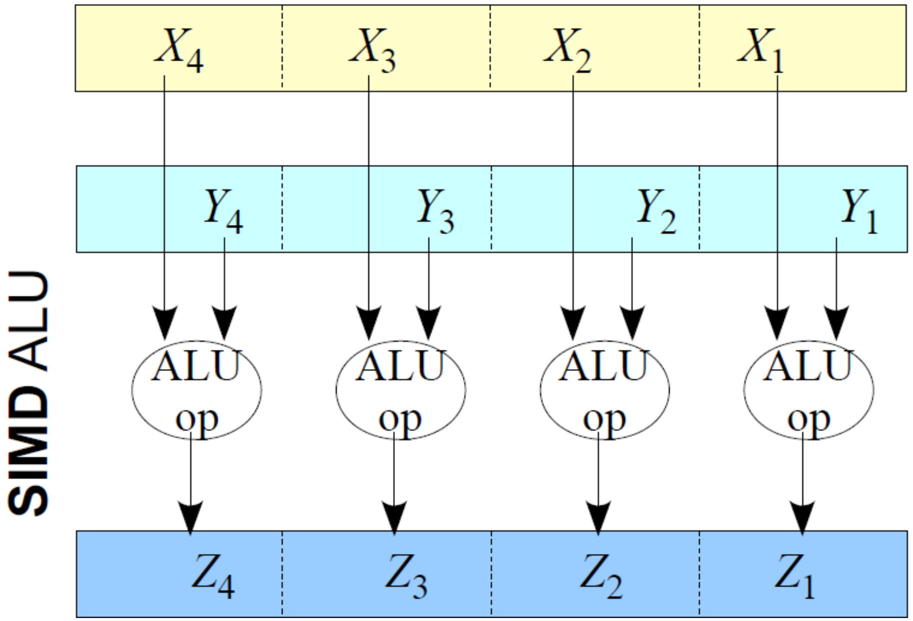
Opcode	Mnemonic	Description
0F 59 /r	MULPS xmm1, xmm2/m128	Multiply packed single-precision floating-point values in xmm2/mem by xmm1.

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];
```



4 pairs of input data operands

4 SIMD results produced simultaneously.
SIMD: Single-Instruction Multiple-Data

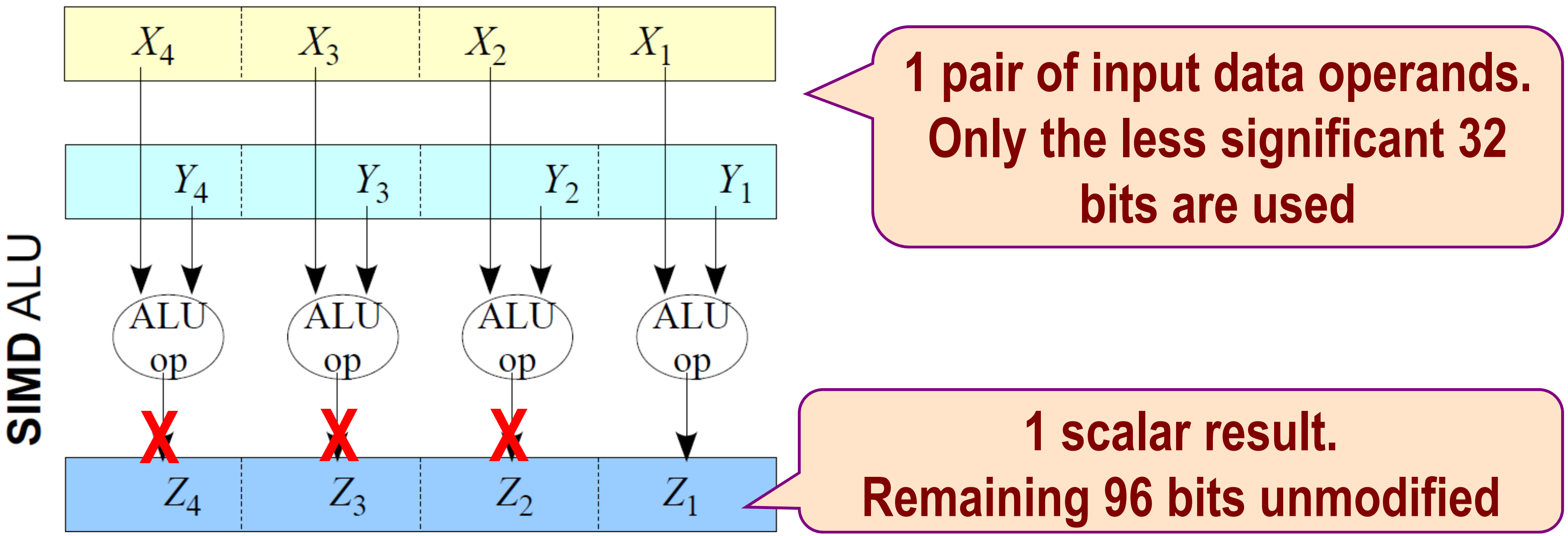
mulss: Multiply Scalar Single-Precision Floating-Point Values

Opcode	Mnemonic	Description
F3 0F 59 /r	MULSS xmm1, xmm2/m32	Multiply the low single-precision floating-point value in xmm2/mem by the low single-precision floating-point value in xmm1.

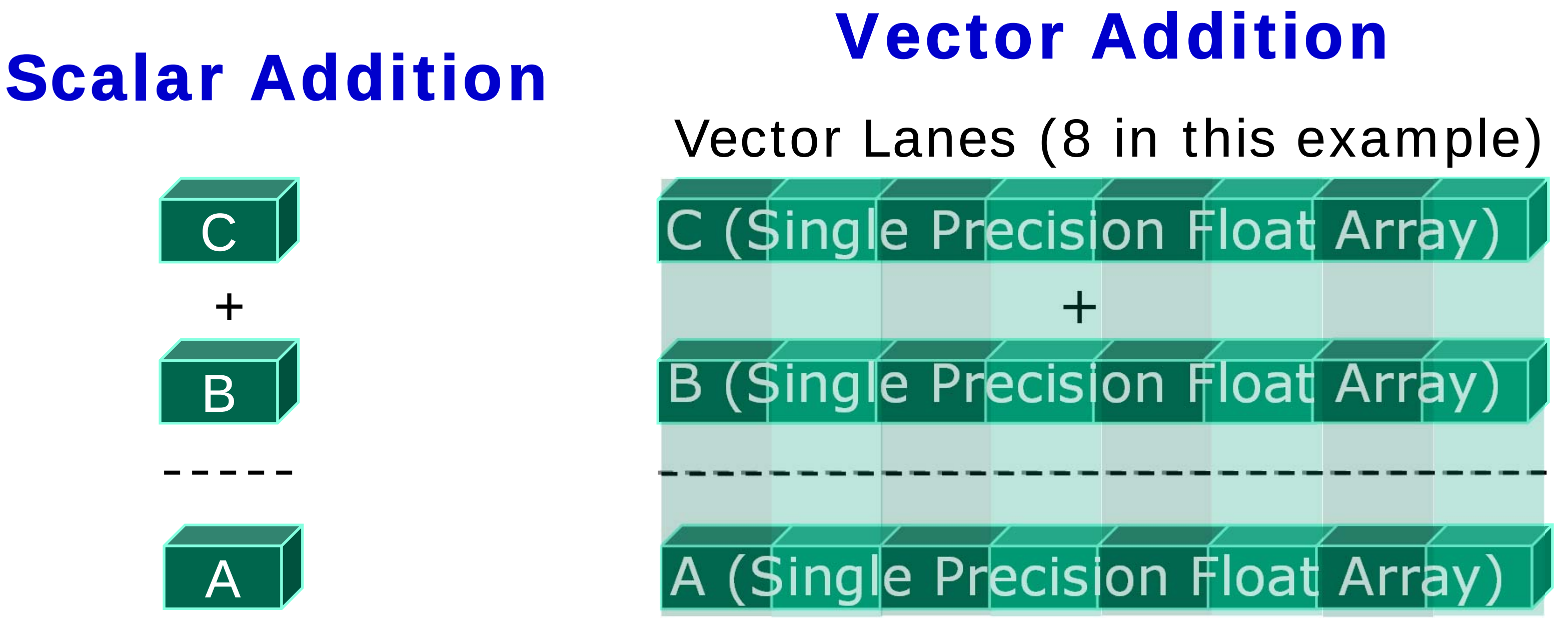
Description
Multiplies the low single-precision floating-point value from the source operand (second operand) by the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1 for an illustration of a scalar single-precision floating-point operation.

Operation

```
Destination[0..31] = Destination[0..31] * Source[0..31];
//Destination[32..127] remains unchanged
```



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

Potential Performance Speedups

Double Precision FP vector width vs
theoretical speedup potential over scalar

128, 256, 512 bit vector divided by **64 bit data type** yields **potential speedups** of **2, 4,** or **8** times

- **Wider vectors** allow for **higher potential performance** gains
- Gains of 4X and 8X within reach using vectorization capability

Ways to Write Vector Code C/C++

Data Level Parallelism with OpenMP* 4.0

Serial Code

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

Let compiler decide if vectorizing is safe

SIMD Pragma/Directive

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

Tell compiler it is safe to vectorize

SIMD-enabled Function

```
#pragma omp declare simd
float foo(float B, float C)
  return B + C;

…

// call foo below
#pragma omp simd
for(i = 0; i < N; i++)
  A[i] = foo(B[i], C[i]);
```

A function inside a vectorized loop must have a vectorized version

```
$ icc -Ofast -openmp file.c -o …
$ gcc -Ofast -fopenmp file.c -o …
```

Requirements for Loop Vectorization

- **Independent iterations:**

  Several loop iterations can be executed in parallel

```
for (i=1; i<N; i++)
  A[i]= A[i-1] + B[i];
```

Not vectorizable:
Iteration i depends on data produced by iteration i-1

Requirements for Loop Vectorization

- **Independent iterations:**

  Several loop iterations can be executed in parallel

- **Consecutive Data Accesses:**

  Consecutive loop iterations access consecutive memory locations

```
for (i=0; i<N; i++)
  A[i][5]= A[i][5] + B[i][2];
```

READ & WRITE DATA no vectorizable:

Cannot read elements A[i][5] and A[i+1][5] from memory (or B[i][2] and B[i+1][2]) with a single SIMD Load instruction

Requirements for Loop Vectorization

- **Independent iterations:**

  Several loop iterations can be executed in parallel

- **Consecutive Data Accesses:**

  Consecutive loop iterations access consecutive memory locations

- **No conditional divergence:**

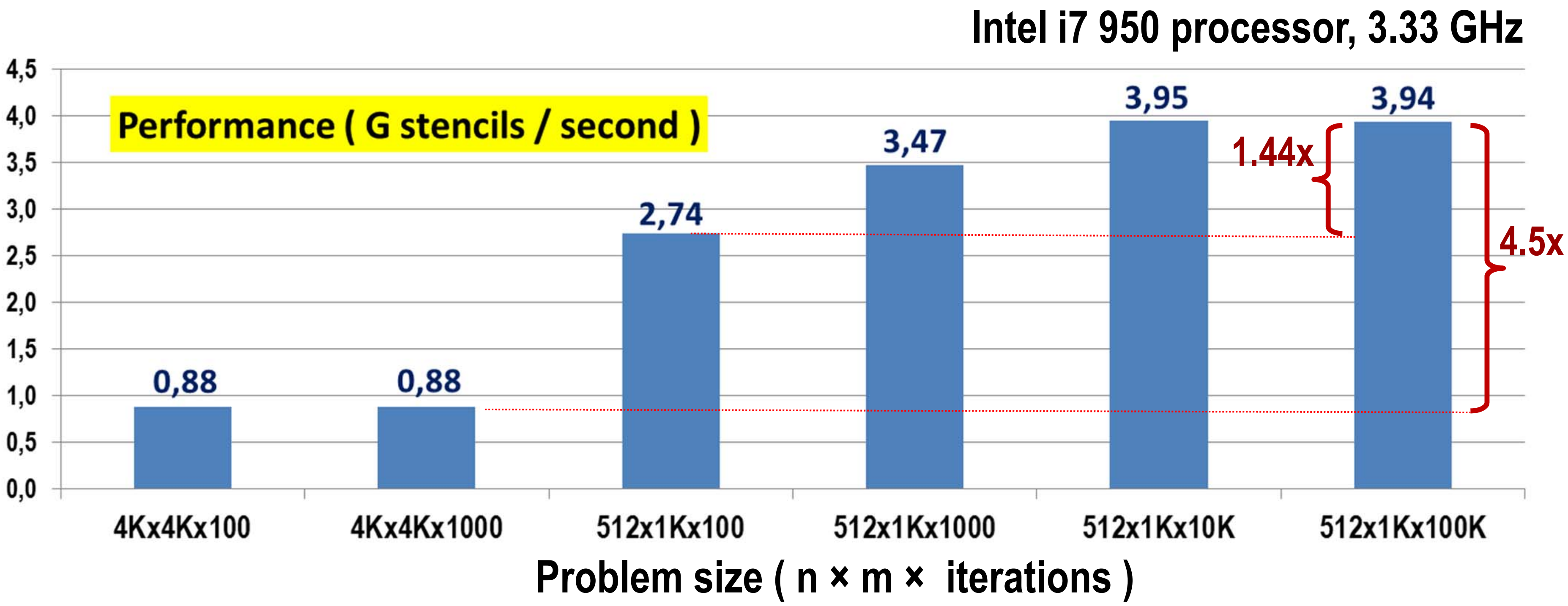
  Conditional statements: some computation done depending on the condition outcome

```
for (i=0; i<N; i++)
  if (A[i]>5) B[i] = A[i]-B[i];
```

Not efficient vectorization:

Masked execution: only some SIMD lanes are active

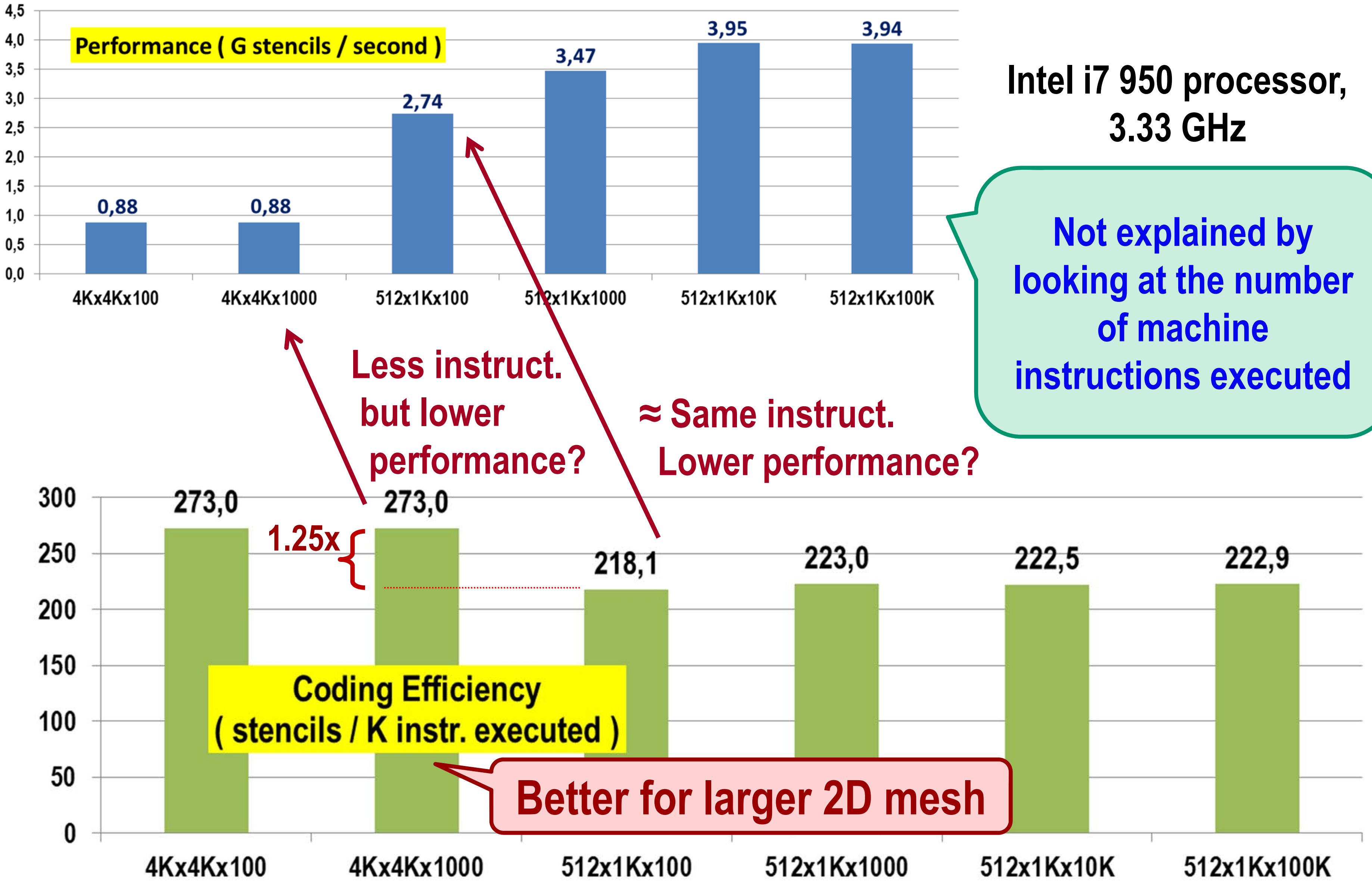
Effect of Problem Size



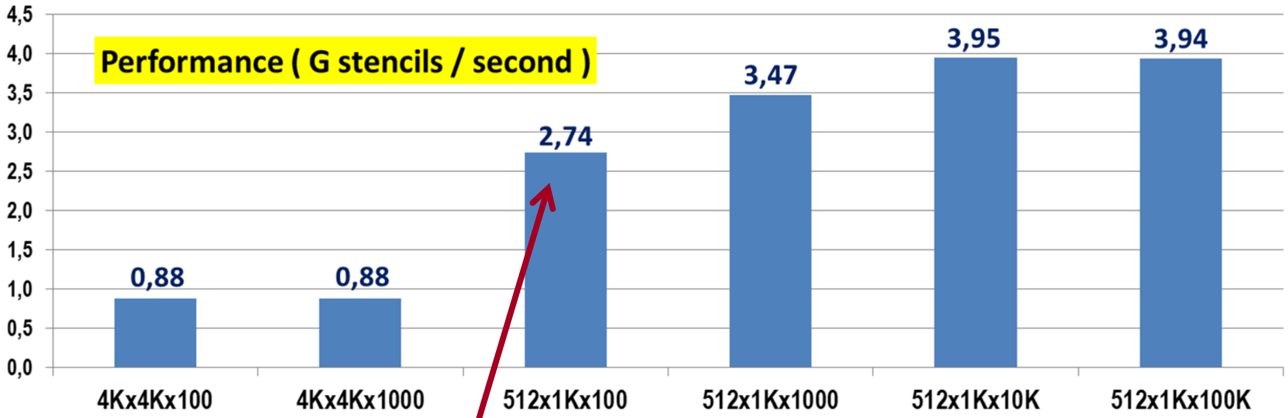
Work / second metric simplifies comparisons for different problem sizes.

How to explain the results?

Explained by Coding Efficiency?



Microarchitecture Throughput?

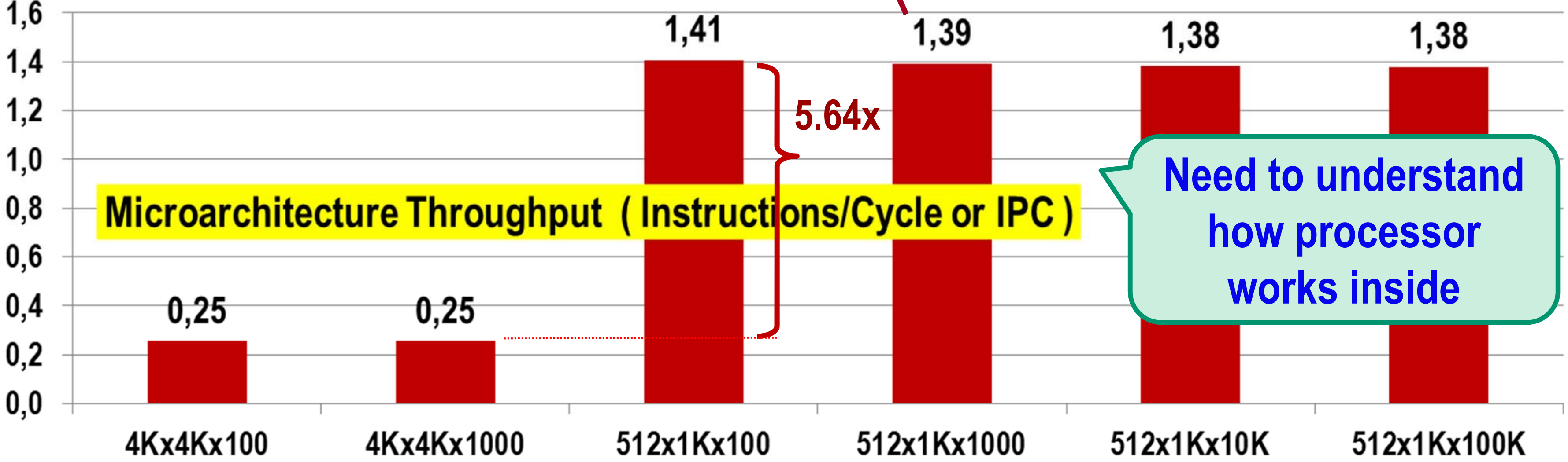


Intel i7 950 processor, 3.33 GHz

Still cannot explain this …

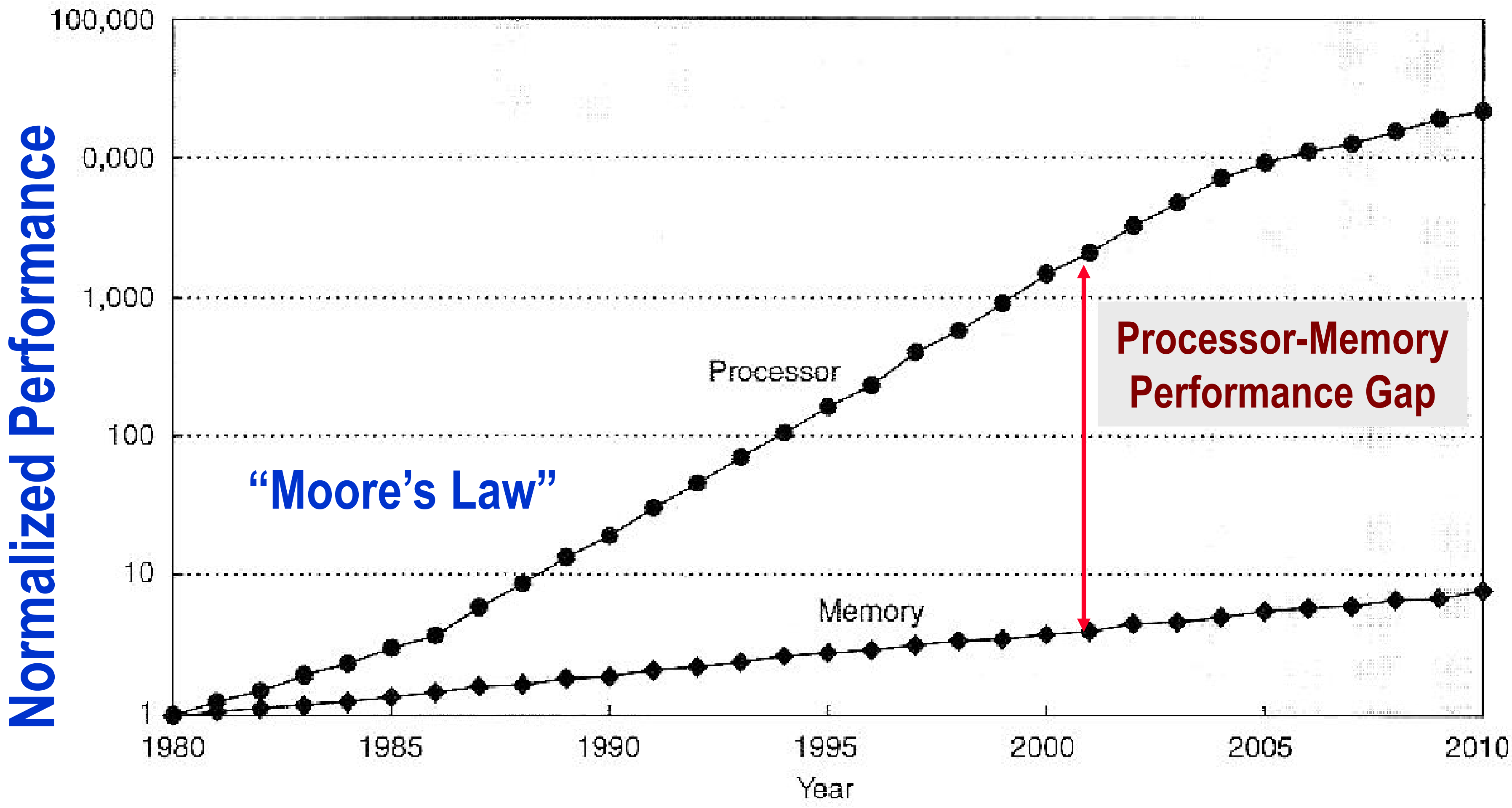
Probably: printf() Input/Output

DO explain this improvement



Need to understand how processor works inside

The CPU-DRAM Gap: the Memory Wall



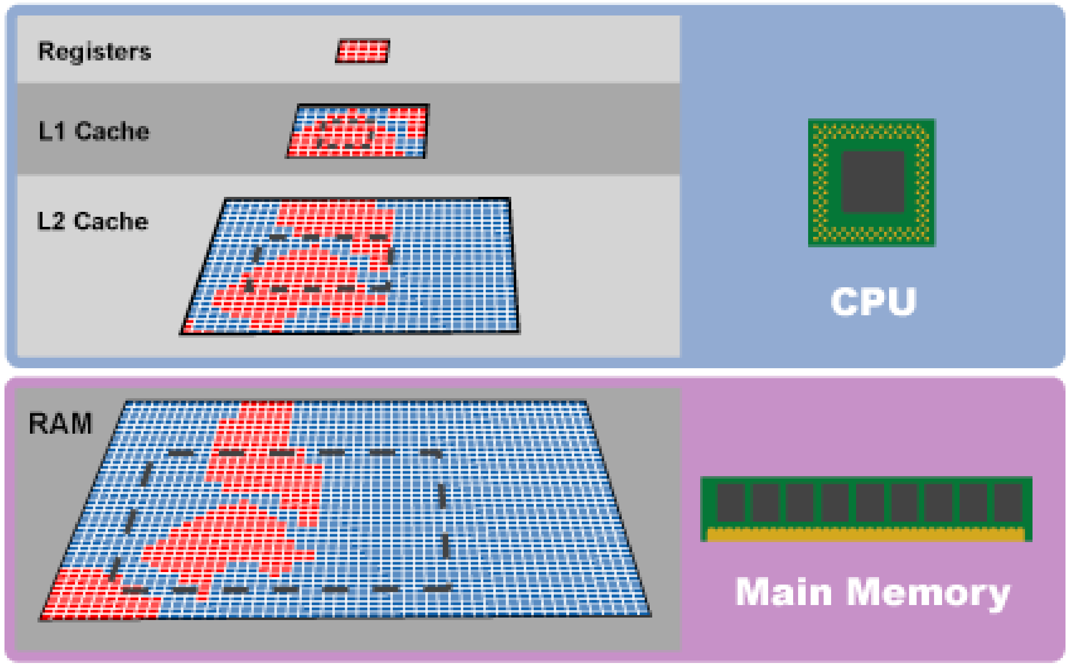
During the latency of a DRAM read operation the processor can execute thousands of instructions

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)

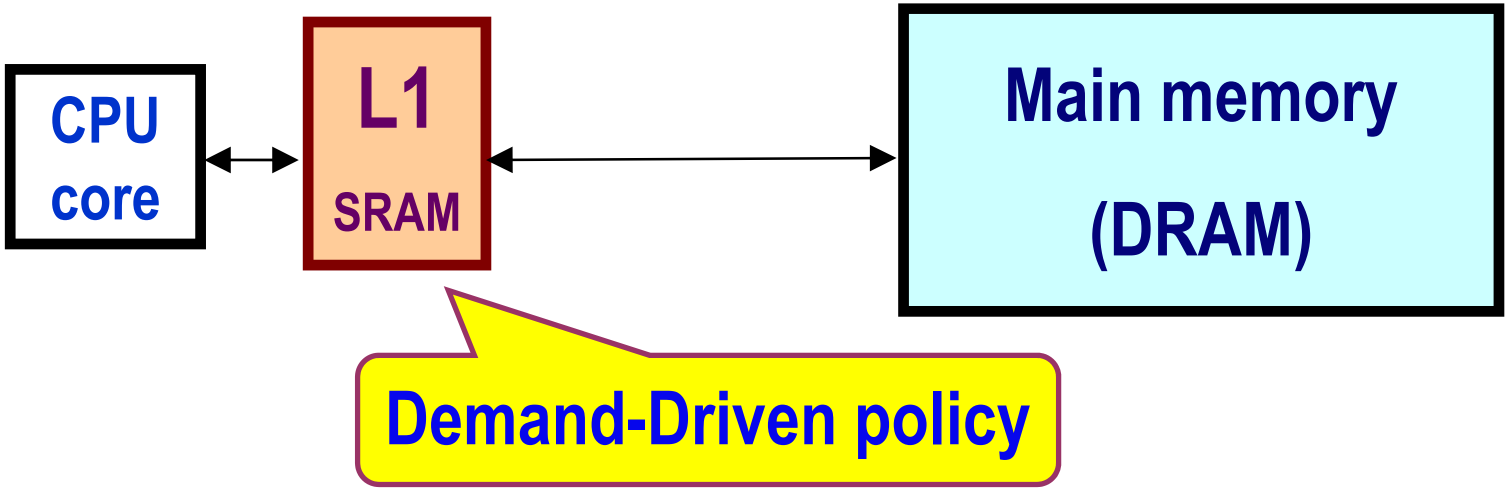


Basic Idea of a Cache

Cache contains **copies** of some memory locations

Memory access does *cache lookup* first (L1 = Level 1).

If desired data is not in the cache, then read the data
from main memory …
and store data in cache (replace some previous data)



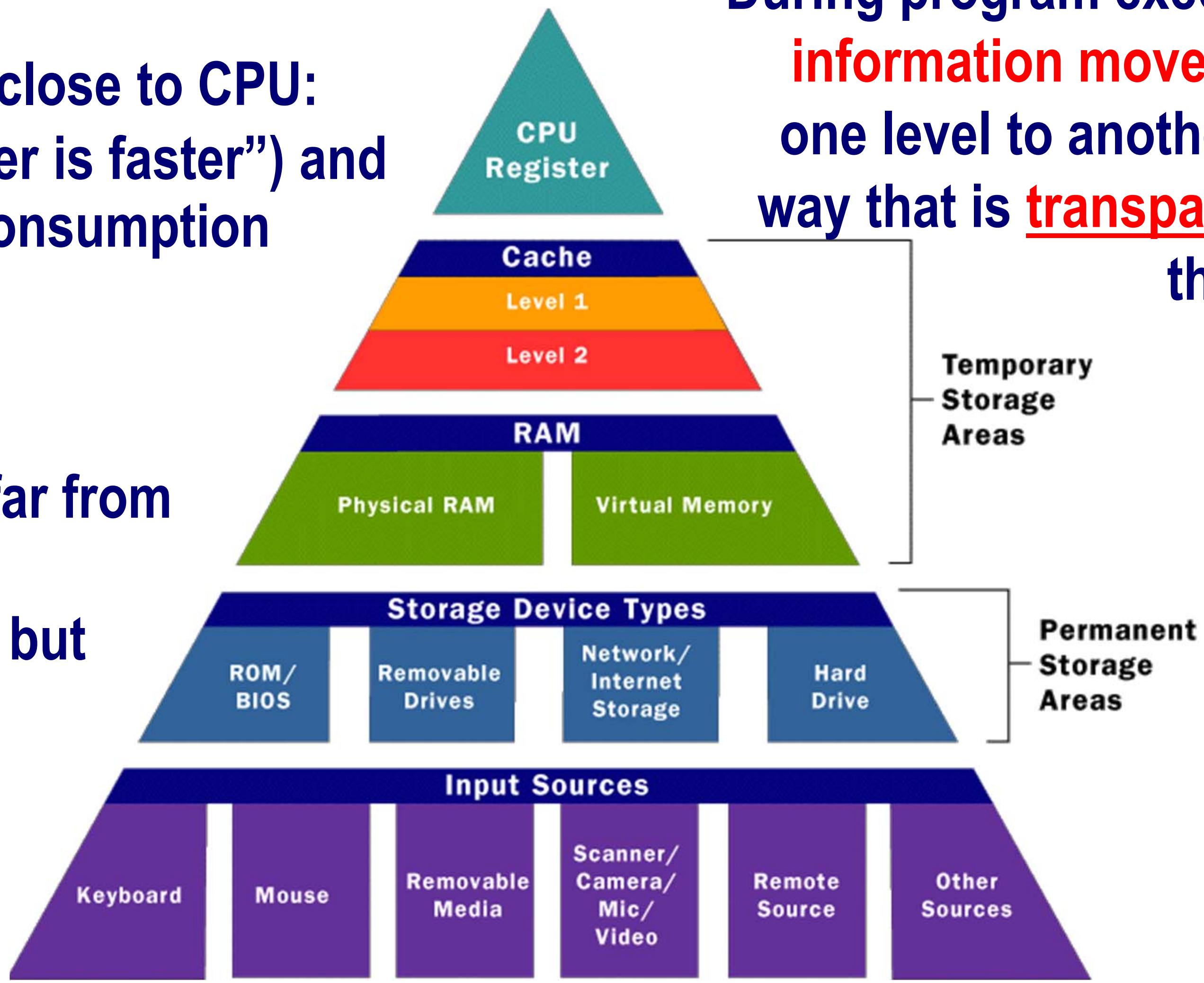
Memory Hierarchy

Illusion of fast, large, cheap, non-volatile, low-energy memory

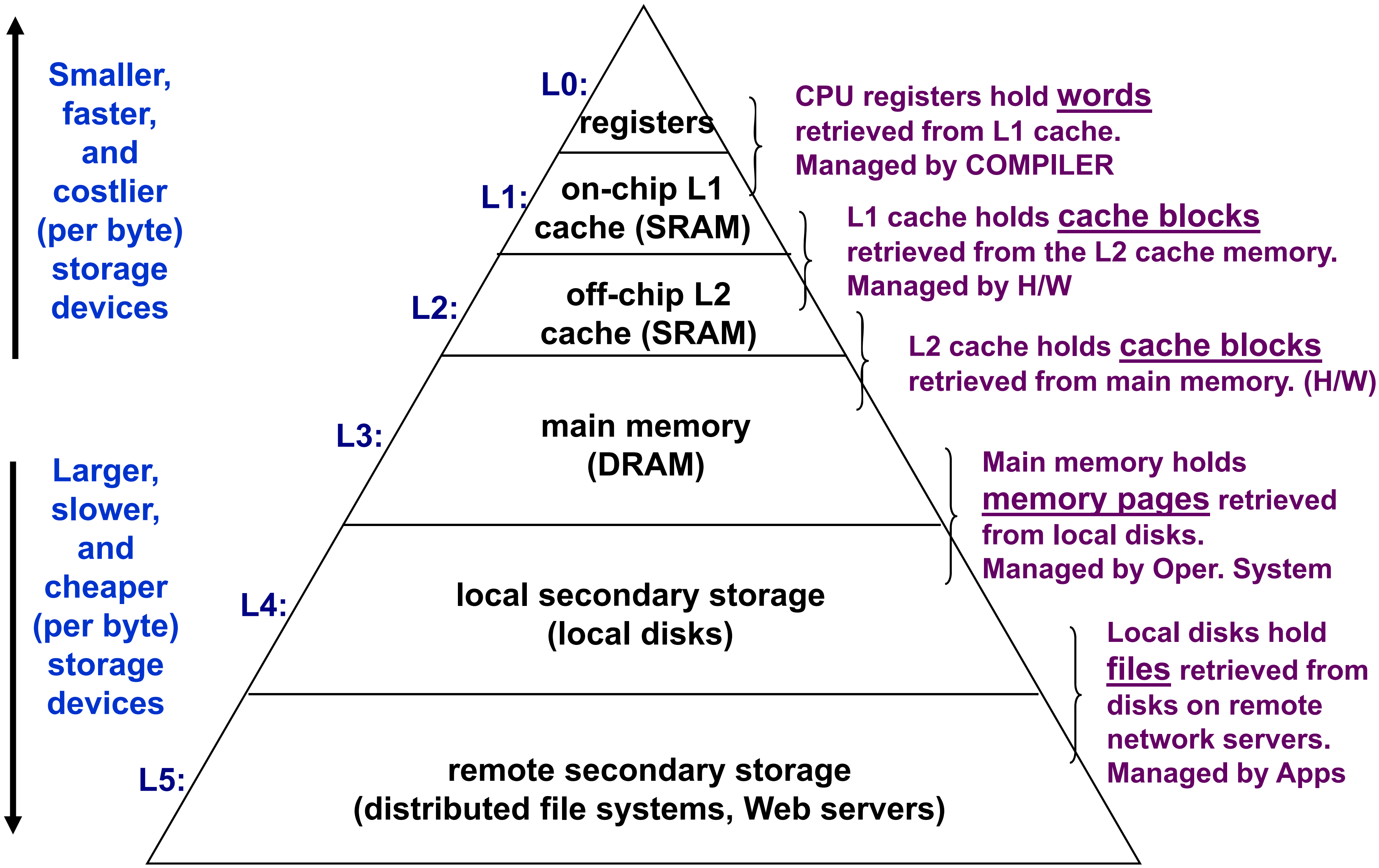
Memory levels close to CPU: **faster**. ("smaller is faster") and lower energy consumption

Memory levels far from CPU: **cheaper** (larger capacity but slower)

During program execution, information moves from one level to another, in a way that is transparent to the user

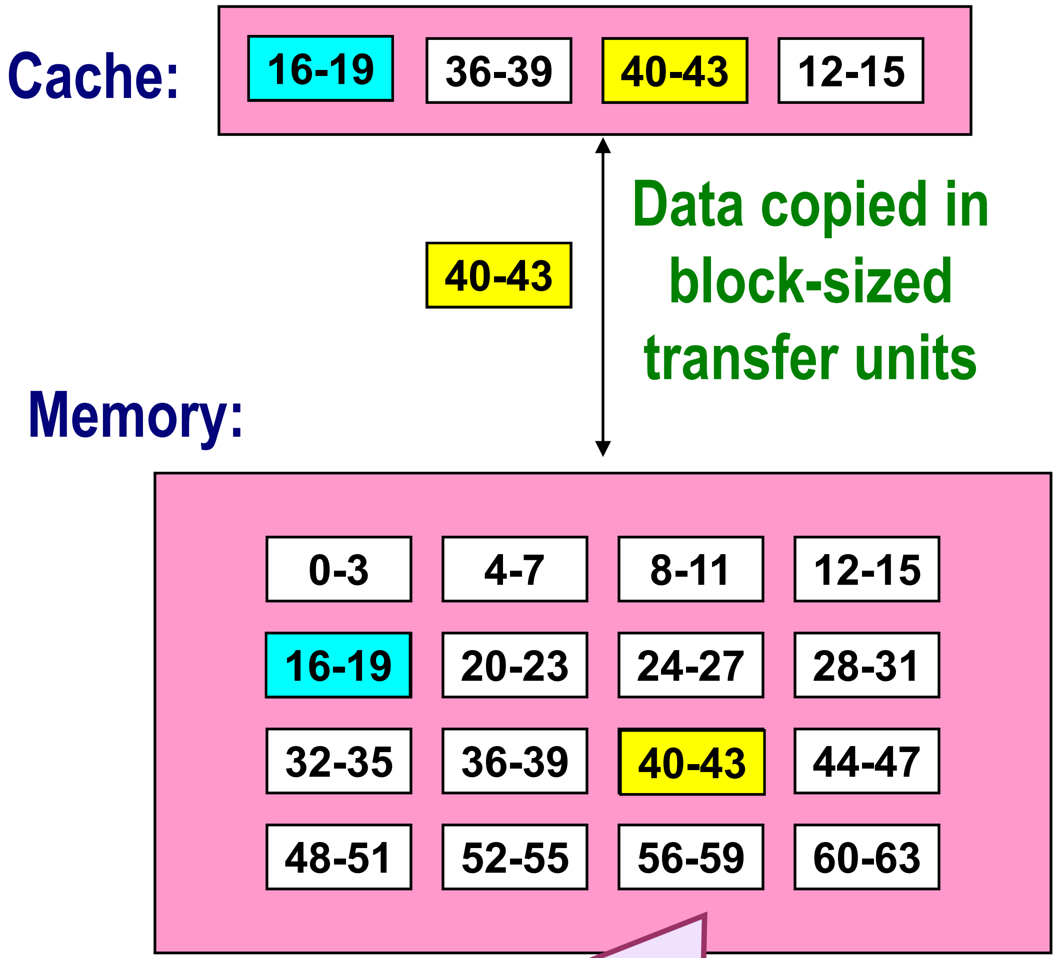


Memory Hierarchy (2)



Cache Organization in Blocks

Smaller, faster, more expensive memory device (caches a subset of the memory blocks into cache lines)



Cache line: physical storage for a block of memory values

Larger, slower, cheaper storage device is partitioned into memory blocks

Memory block: logical partition of consecutive memory locations

Blocks are aligned into memory

Cache Organization in Blocks (2)

Why Blocks?
Exploit Spatial Locality

Smaller, faster, more expensive memory device (caches a subset of the memory blocks into <u>cache lines</u>)

Ca

Data copied in

Memory

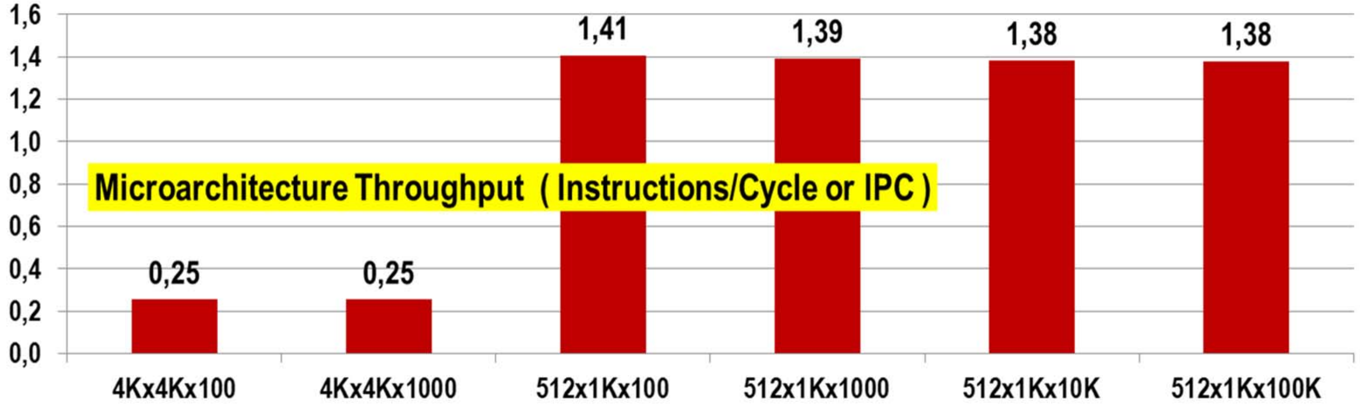
Blocks a

<u>Rule of Thumb for PERFORMANCE:</u>

When the program requests a new data block from memory, most of the data in the block should be used (maybe several times) in a relatively short period of time

This naturally occurs when data is read from consecutive positions in an array

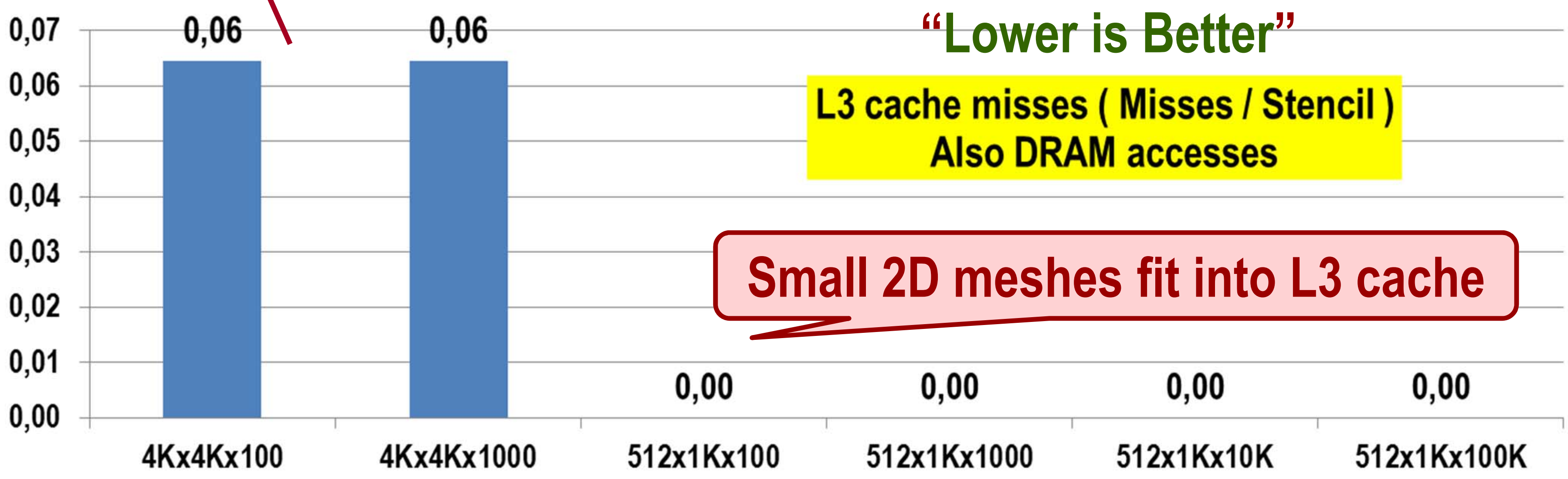
Measure Memory Cache Performance



Intel i7 950 processor, 3.33 GHz

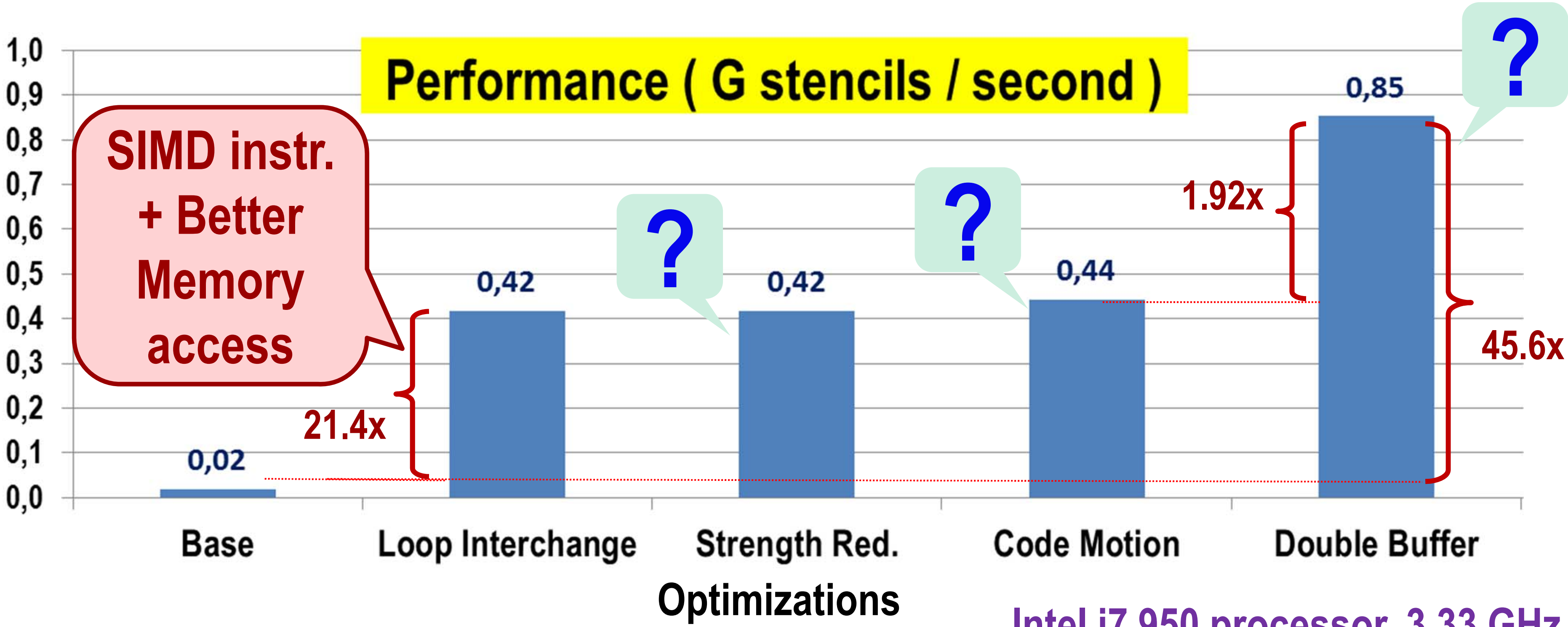
Qualitative Explanation

More accesses to DRAM lowers the average IPC, because instructions must wait for too long



Comparing Performance: Speedup

Problem Size = 2K × 2K matrix, 100 convergence iterations

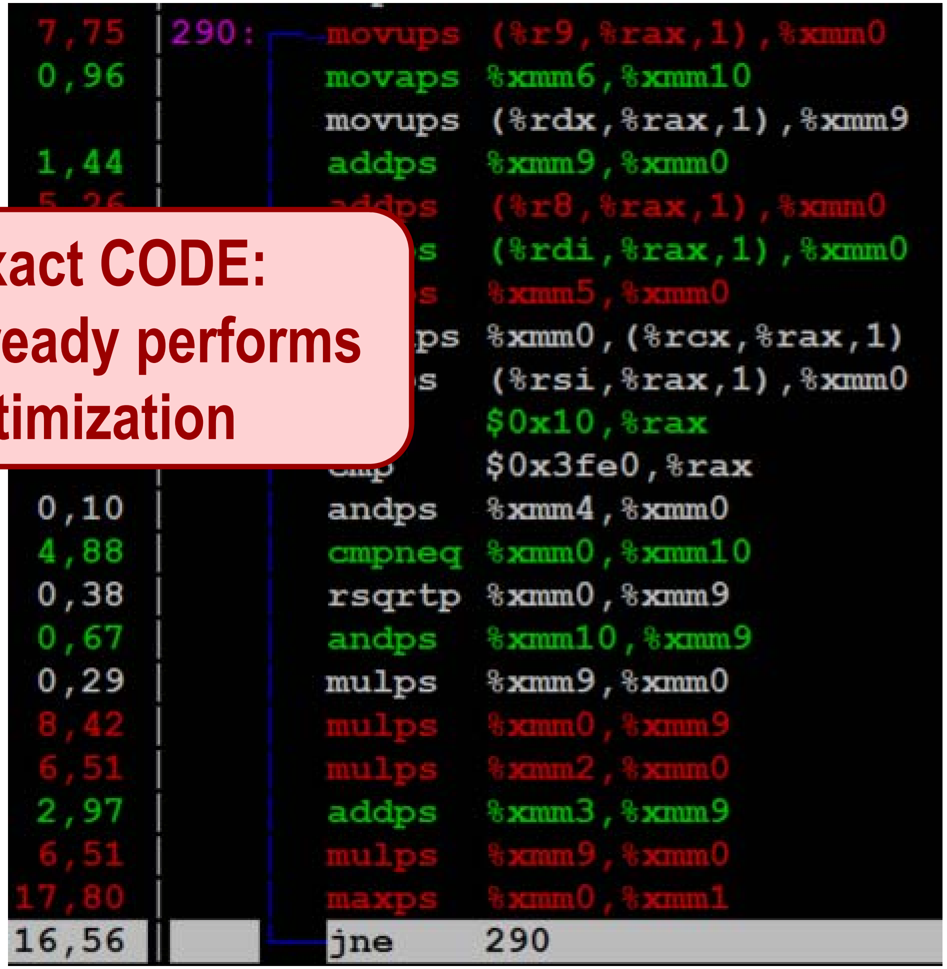


Intel i7 950 processor, 3.33 GHz

Loop interchange:	very large improvement (speedup is 21.4x)
Double Buffer:	almost doubles performance (1.92x)
Strength Red. & Code motion:	very small or no improvement

Assembly: analysis & comparison (2)

gcc –Ofast (v 6.1)



Same Exact CODE:
Compiler already performs this optimization

Loop interchange

Strength reduction

Assembly: analysis & comparison (3)

```
 7,75 |290: movups  (%r9,%rax,1),%xmm0
 0,96 |     movaps  %xmm6,%xmm10
      |     movups  (%rdx,%rax,1),%xmm9
 1,44 |     addps   %xmm9,%xmm0
 5,26 |     addps   (%r8,%rax,1),%xmm0
 0,67 |     addps   (%rdi,%rax,1),%xmm0
12,54 |     mulps   %xmm5,%xmm0
 0,29 |     movaps  %xmm0,(%rcx,%rax,1)
 0,10 |     subps   (%rsi,%rax,1),%xmm0
 4,98 |     add     $0x10,%rax
      |     cmp     $0x3fe0,%rax
 0,10 |     andps   %xmm4,%xmm0
 4,88 |     cmpneq  %xmm0,%xmm10
 0,38 |     rsqrtp  %xmm0,%xmm9
 0,67 |     andps   %xmm10,%xmm9
 0,29 |     mulps   %xmm9,%xmm0
 8,42 |     mulps   %xmm0,%xmm9
 6,51 |     mulps   %xmm2,%xmm0
 2,97 |     addps   %xmm3,%xmm9
 6,51 |     mulps   %xmm9,%xmm0
17,80 |     maxps   %xmm0,%xmm1
16,56 ||    jne     290
```

Code Motion: remove computation of square root from inner loop

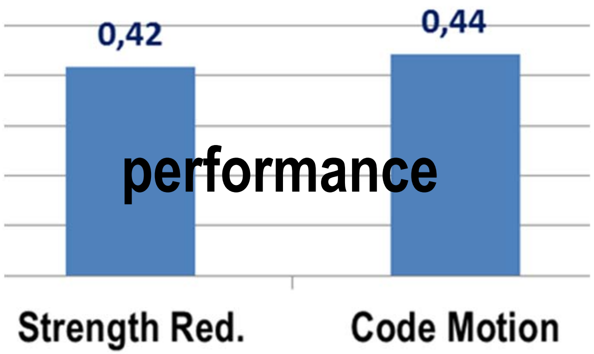
Strength reduction

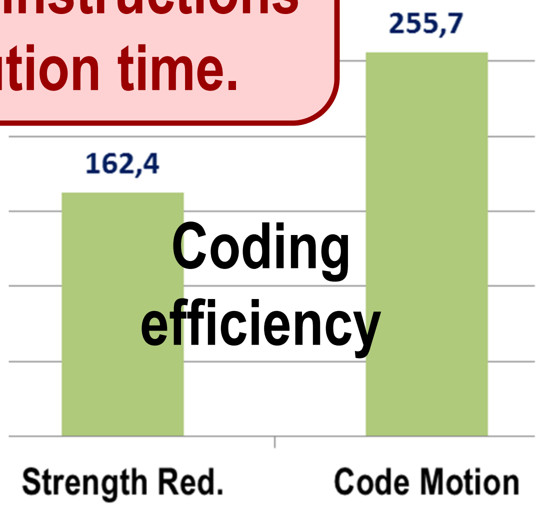
```
 7,53 |278: movups  (%r9,%rax,1),%xmm0
 1,06 |     movups  (%rdx,%rax,1),%xmm2
 6,71 |     addps   %xmm2,%xmm0
 1,18 |     addps   (%r8,%rax,1),%xmm0
 9,65 |     addps   (%rdi,%rax,1),%xmm0
33,29 |     mulps   %xmm4,%xmm0
12,00 |     movaps  %xmm0,(%rcx,%rax,1)
 2,12 |     subps   (%rsi,%rax,1),%xmm0
 9,53 |     add     $0x10,%rax
      |     cmp     $0x3fe0,%rax
      |     andps   %xmm3,%xmm0
 2,82 |     maxps   %xmm0,%xmm1
13,65 ||    jne     278
```

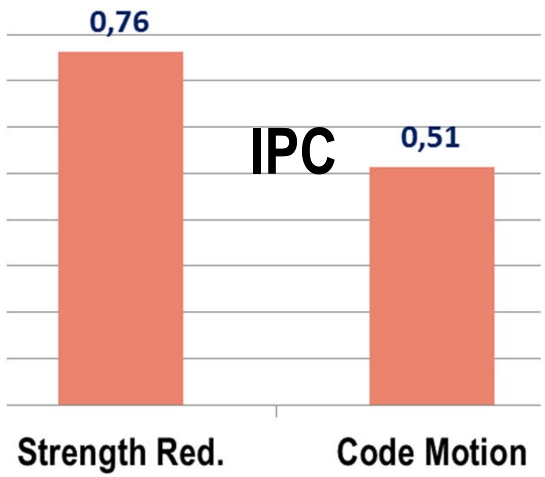
gcc –Ofast (v 6.1)

Performance bottleneck is waiting for data in memory: reducing instructions does not improve execution time.

Why IPC degrades?







Assembly: analysis & comparison (3)

```
Samples: 154K of event 'cycles', Event count (approx.): 128527894268
Overhead  Command  Shared Object       Symbol
  53,27%  Lsqrt    Lsqrt               [.] main
  45,82%  Lsqrt    libc-2.17.so        [.] __memcpy_ssse3_back
   0,04%  Lsqrt    [kernel.kallsyms]   [k] update_wall_time
   0,04%  Lsqrt    [kernel.kallsyms]   [k] apic_timer_interrupt
   0,04%  Lsqrt    [kernel.kallsyms]   [k] rcu_check_callbacks
   0,03%  Lsqrt    [kernel.kallsyms]   [k] trigger_load_balance
```

Code Motion

gcc –Ofast (v 6.1)

```
Samples: 94K of event 'cycles', Event count (approx.): 76992055233
Overhead  Command  Shared Object       Symbol
  99,21%  L2buff   L2buff              [.] main
   0,05%  L2buff   [kernel.kallsyms]   [k] apic_time
   0,03%  L2buff   [kernel.kallsyms]   [k] rcu_check
   0,03%  L2buff   [kernel.kallsyms]   [k] clear_pag
   0,03%  L2buff   [kernel.kallsyms]   [k] task_tick
   0,02%  L2buff   [kernel.kallsyms]   [k] trigger_l
   0,02%  L2buff   libc-2.17.so        [.] __memset_
```

Double Buffer

The few instructions doing the memory copy were responsible of most of the cycles

Why IPC improves?