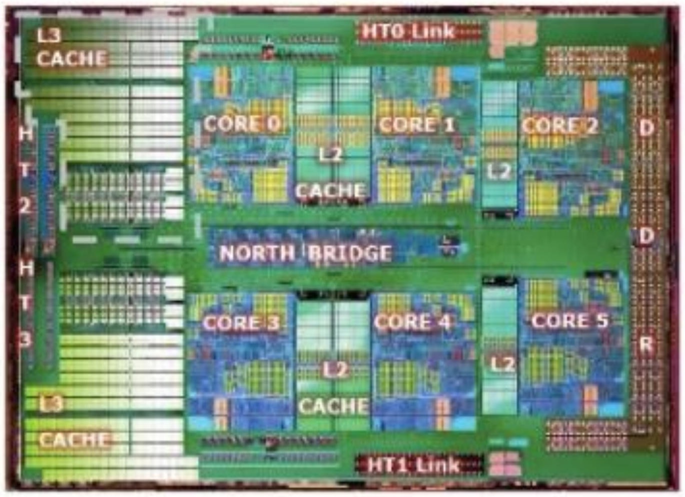
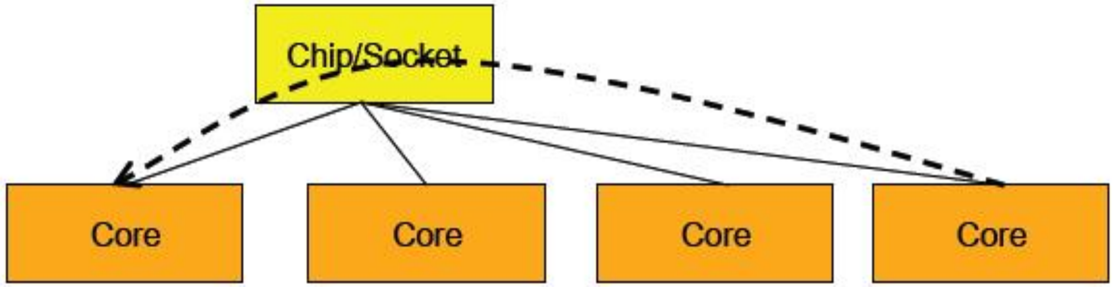
程序性能优化
Part I

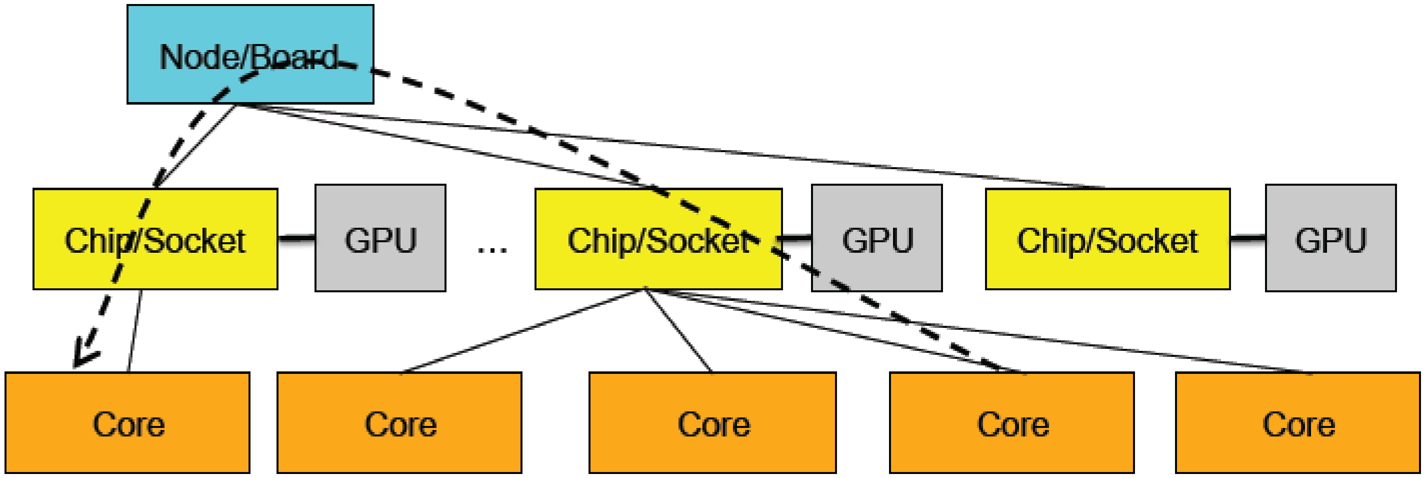
处理器结构


L3 CACHE
HT0 Link
CORE 0
L2 CACHE
CORE 1
CORE 2
L2
D
H T 2
NORTH BRIDGE
D
H T 3
CORE 3
CORE 4
CORE 5
L2 CACHE
L2
R
L3 CACHE
HT1 Link


Chip/Socket
Core
Core
Core
Core

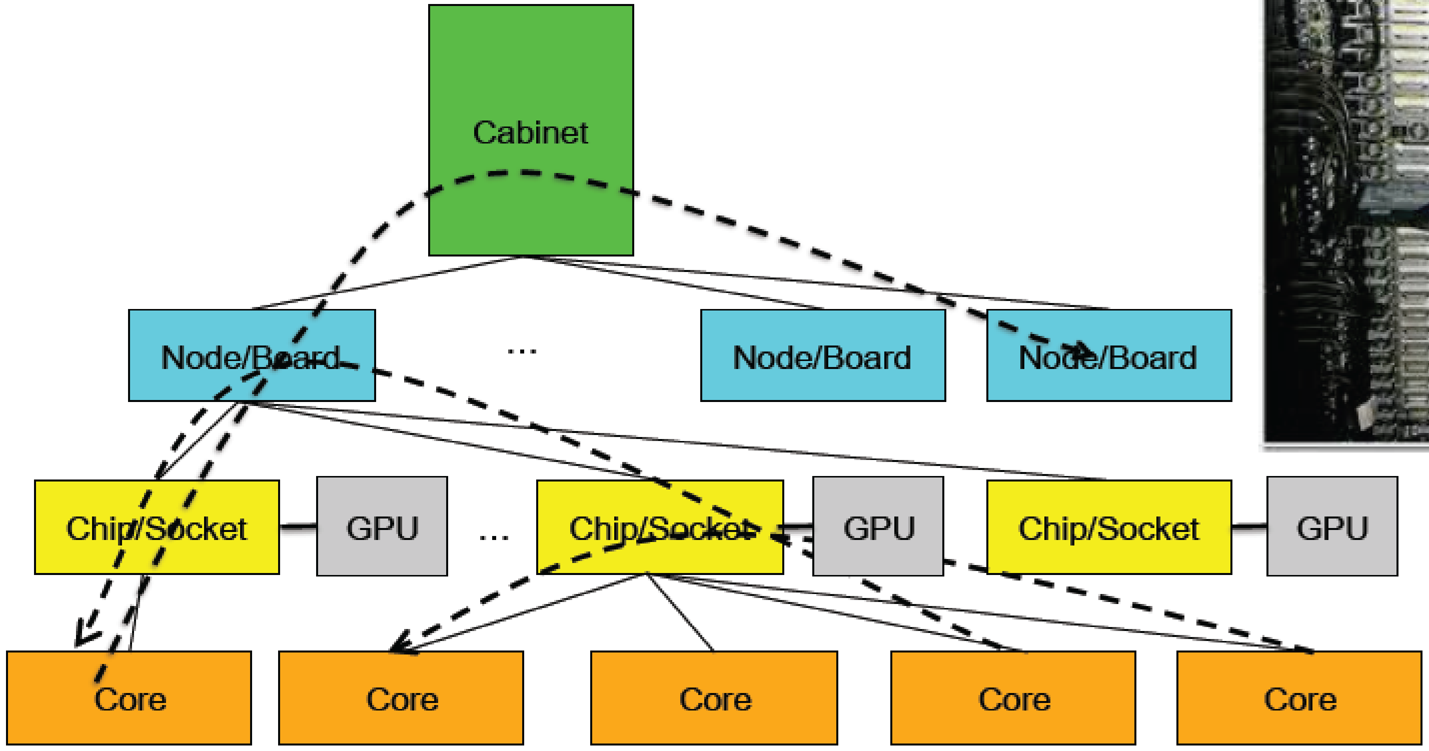
节点结构

NVIDIA Tesla M1060 GPU Card
NVIDIA Tesla M1060 GPU Card

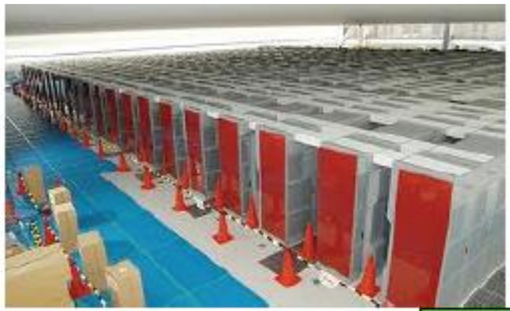

Node/Board
Chip/Socket
GPU
...
Chip/Socket
GPU
Chip/Socket
GPU
Core
Core
Core
Core
Core

机柜结构

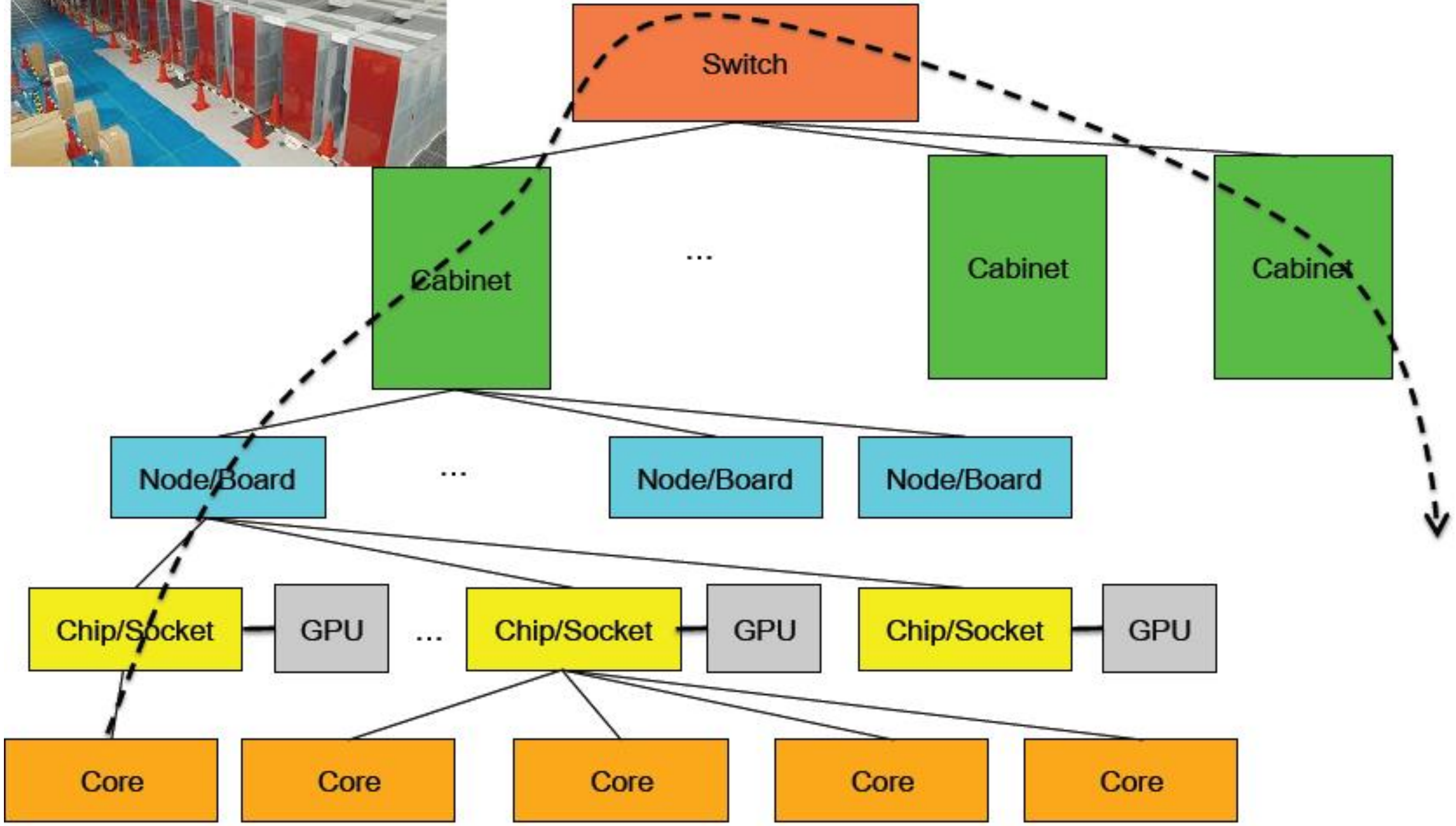
Shared memory programming between processes on a board and
a combination of shared memory and distributed memory programming
between nodes and cabinets



系统结构



Combination of shared memory and distributed memory programming



从程序优化角度，我们的分类

- Tuning for In-core programming

- Tuning for Parallel programming

从程序优化角度，我们的课程安排

- Week 1: Tuning for In-core programming
- Week 2: Tools: Intel compiler and Vtune
- Week 3: Tuning for Parallel programming
- Week 4: Show case

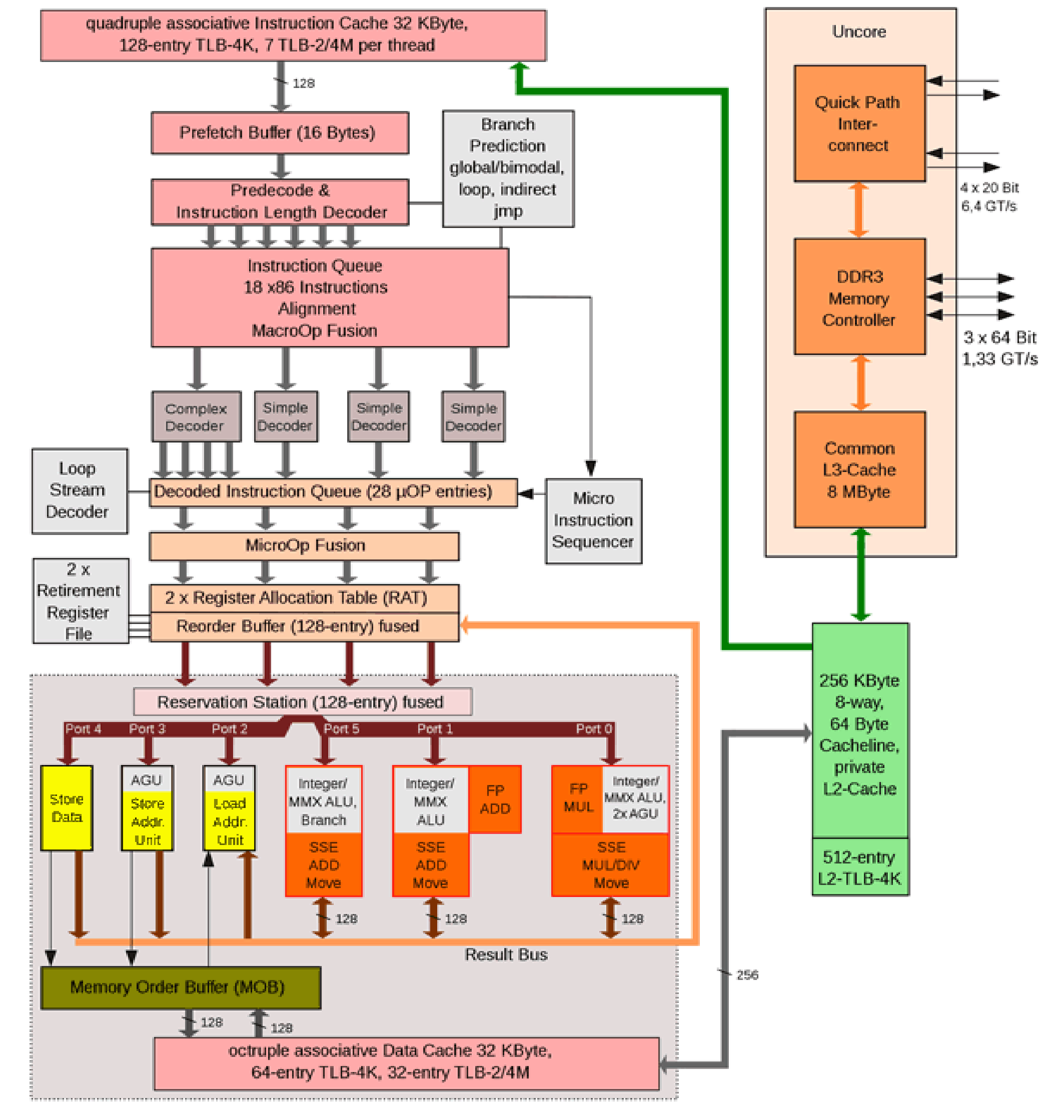
Helping the Compiler for Performance

Outline

- **Behind the compiler – the architecture basis**
- Tricks Compilers Play
  - Scalar Optimizations
  - Loop Optimizations
  - Inlining
- Where is the problem and how to do
- Tricks You Can Play with Compilers

现代处理体系结构

Intel Nehalem microarchitecture


quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octruple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

GT/s: gigatransfers per second

- Registers
- Cache
- Execution Units (向量)

First thing to know: What Is ILP?

Instruction-Level Parallelism (ILP) is a set of techniques for **executing multiple instructions at the same time within the same CPU core.**

(Note that ILP has **nothing to do with multicore**.)

The problem: A CPU core has lots of circuitry, and at any given time, most of it is idle, which is wasteful.

The solution: Have different parts of the CPU core work on different operations at the same time: If the CPU core has the ability to work on 10 operations at a time, then the program can, in principle, run as much as 10 times as fast (although in practice, not quite so much).

Why You Shouldn’t Panic

In general, the compiler and the CPU will do most of the heavy lifting for instruction-level parallelism.

BUT:

You need to be aware of ILP, because how your code is structured affects how much ILP the compiler and the CPU can give you.

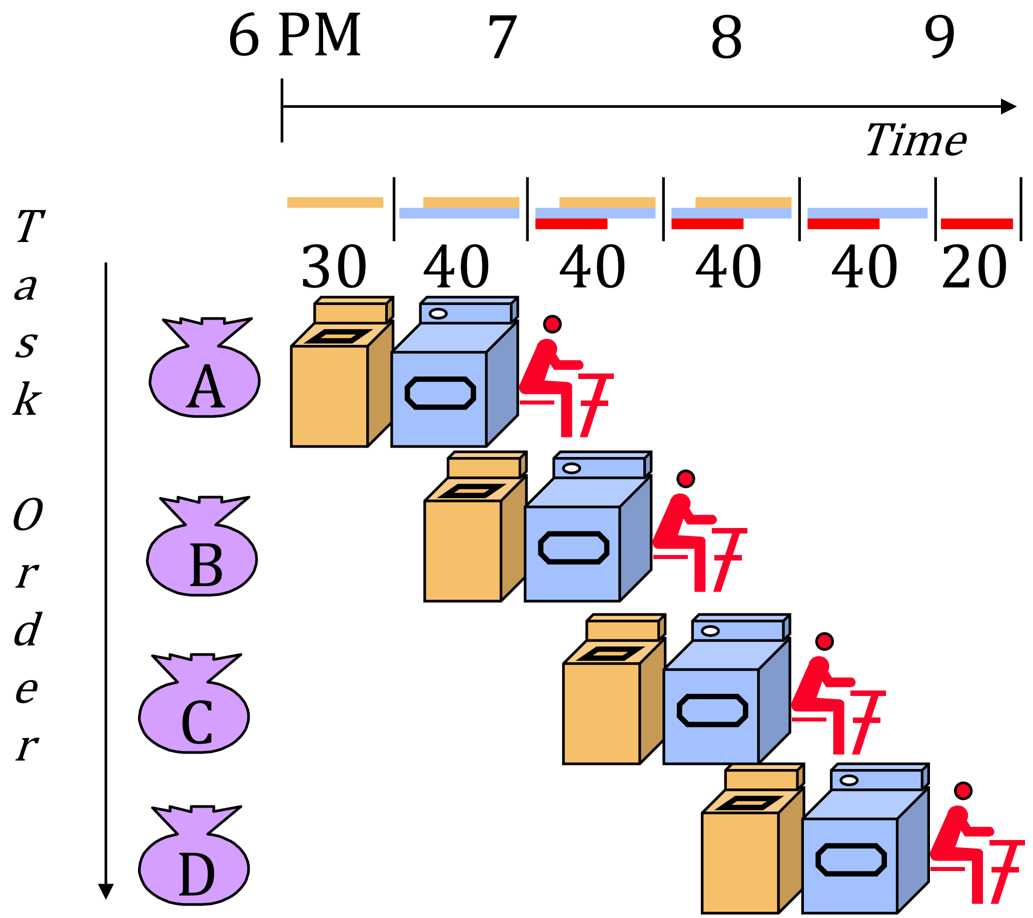
Kinds of ILP

- ***Superscalar***: Perform multiple operations at the same time (for example, simultaneously perform an add, a multiply and a load).
- ***Pipeline***: Start performing an operation on one piece of data while finishing the same operation on another piece of data – perform different ***stages*** of the same operation on different sets of operands at the same time (like an assembly line).
- ***Superpipeline***: A combination of superscalar and pipelining – perform multiple pipelined operations at the same time.
- ***Vector***: Load multiple pieces of data into special registers and perform the same operation on all of them at the same time.

Pipelining

Dave Patterson's Laundry example: 4 people doing laundry

wash (30 min) + dry (40 min) + fold (20 min) = 90 min Latency

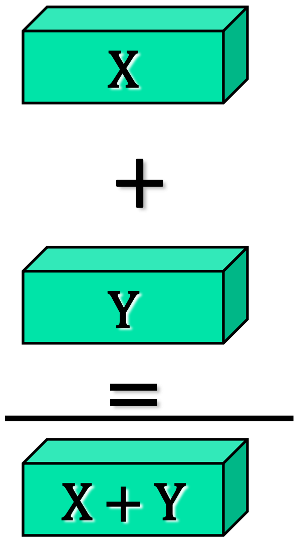


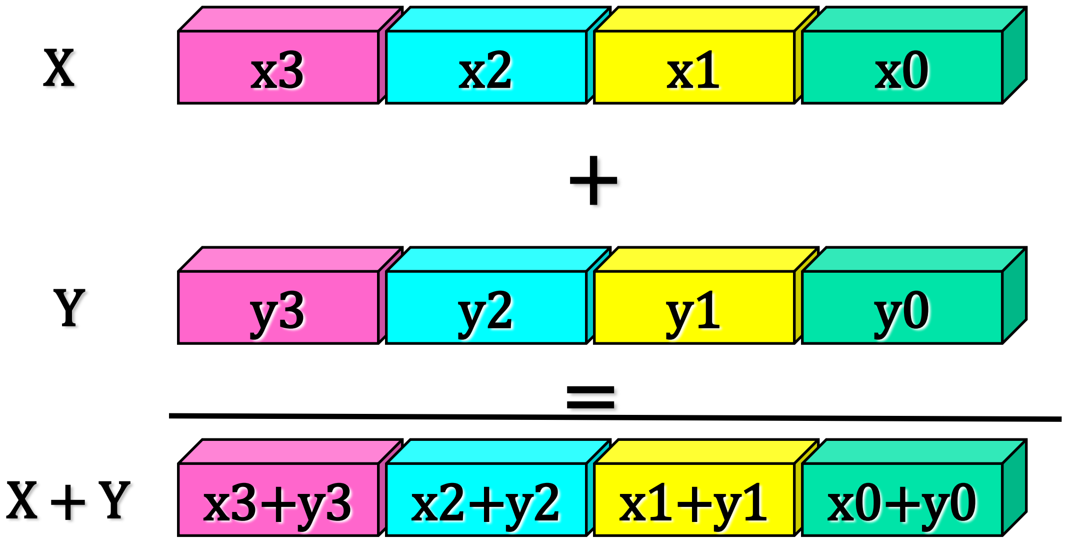
- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hours
- Bandwidth = loads/hour
- BW = 4/6 l/h w/o pipelining
- BW = 4/3.5 l/h w pipelining
- BW <= 1.5 l/h w pipelining, more total loads
- **Pipelining helps bandwidth but not latency (90 min)**
- **Bandwidth limited by slowest pipeline stage**
- **Potential speedup = Number pipe stages**

Vectorization

- Scalar processing
  - traditional mode
  - one instruction produces one result

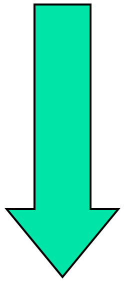
- SIMD processing
  - with SSE(2,3,4)
  - one instruction produces multiple results





Second thing to know: The Storage Hierarchy

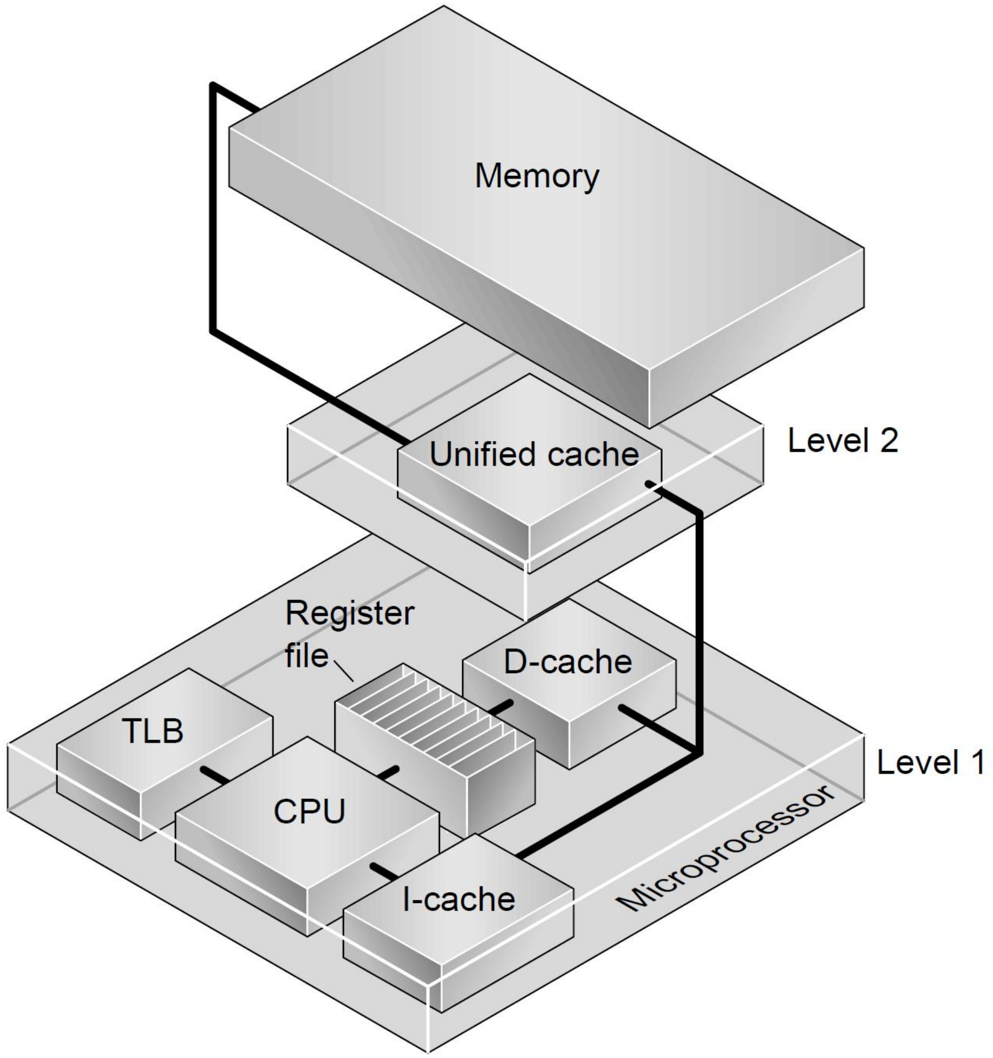
Fast, expensive, few



Slow, cheap, a lot

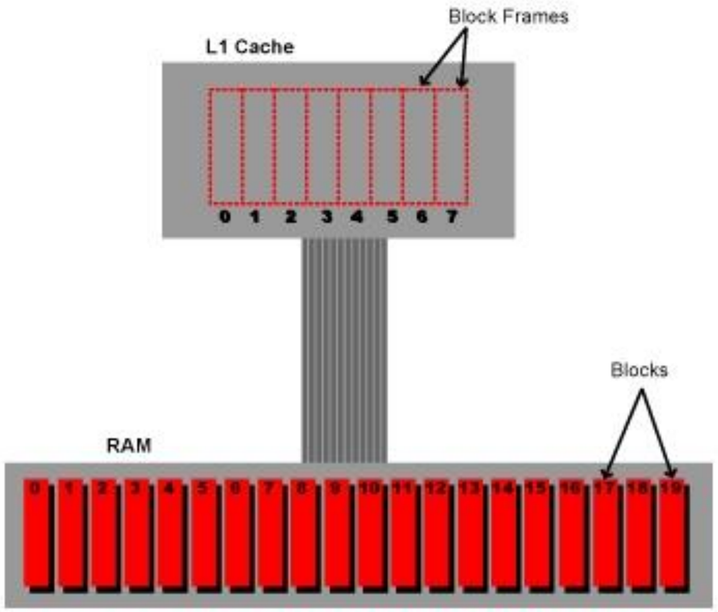
- Registers
- Cache memory
- Main memory (RAM)
- Hard disk
- Removable media (CD, DVD etc)
- Internet

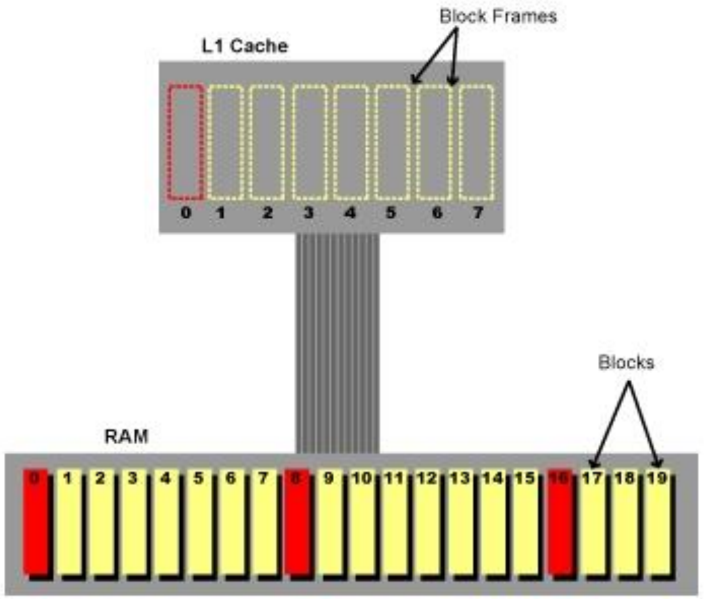
Memory Hierarchy

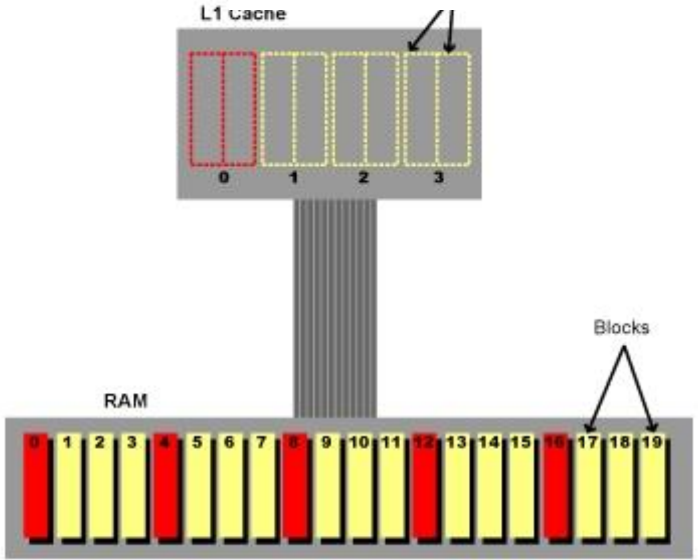


- For a programmer, we can not control the caches. But we should aware of cache.

Cache


L1 Cache
Block Frames
0 1 2 3 4 5 6 7
Blocks
RAM
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19


L1 Cache
Block Frames
0 1 2 3 4 5 6 7
Blocks
RAM
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19


L1 Cache
0 1 2 3
Blocks
RAM
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

A Laptop

Dell Latitude Z600[4]

- Intel Core2 Duo SU9600
  1.6 GHz w/3 MB L2 Cache
- 4 GB 1066 MHz DDR3 SDRAM
- 256 GB SSD Hard Drive
- DVD±RW/CD-RW Drive (8x)
- 1 Gbps Ethernet Adapter

Storage Speed, Size, Cost

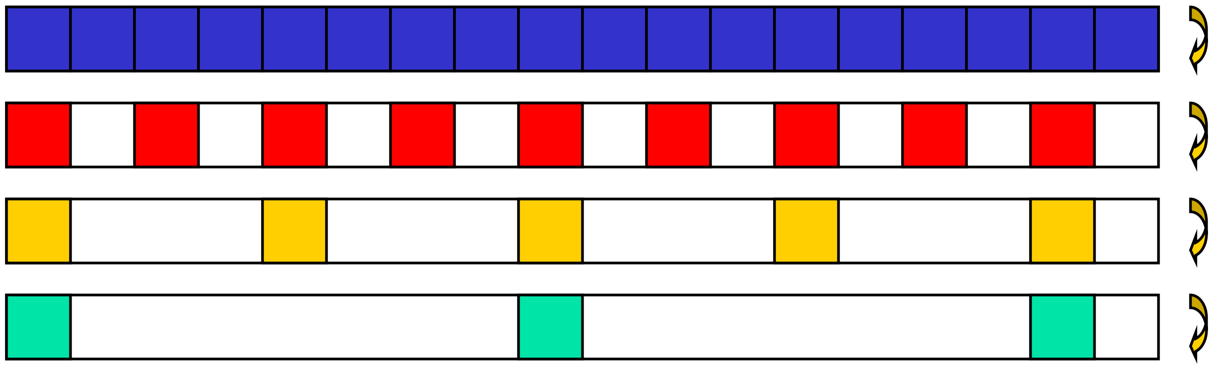
Laptop	Registers (Intel Core2 Duo 1.6 GHz)	Cache Memory (L2)	Main Memory (1066MHz DDR3 SDRAM)	Hard Drive (SSD)	Ethernet (1000 Mbps)	DVD±R (16x)	Phone Modem (56 Kbps)
Speed (MB/sec) [peak]	314,573 (12,800 MFLOP/s*)	27,276	4500	250	125	22	0.007
Size (MB)	464 bytes**	3	4096	256,000	unlimited	unlimited	unlimited
Cost ($/MB)	–	$285	$0.03	$0.002	charged per month (typically)	$0.00005	charged per month (typically)

* MFLOP/s: millions of floating point operations per second

** 16 64-bit general purpose registers, 8 80-bit floating point registers, 16 128-bit floating point vector registers

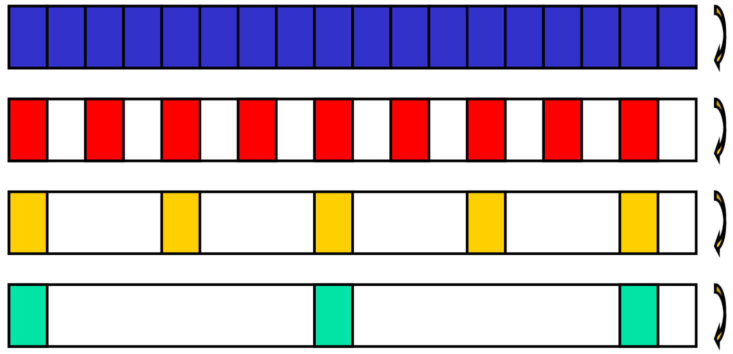
Experimental Study of Memory (Membench)

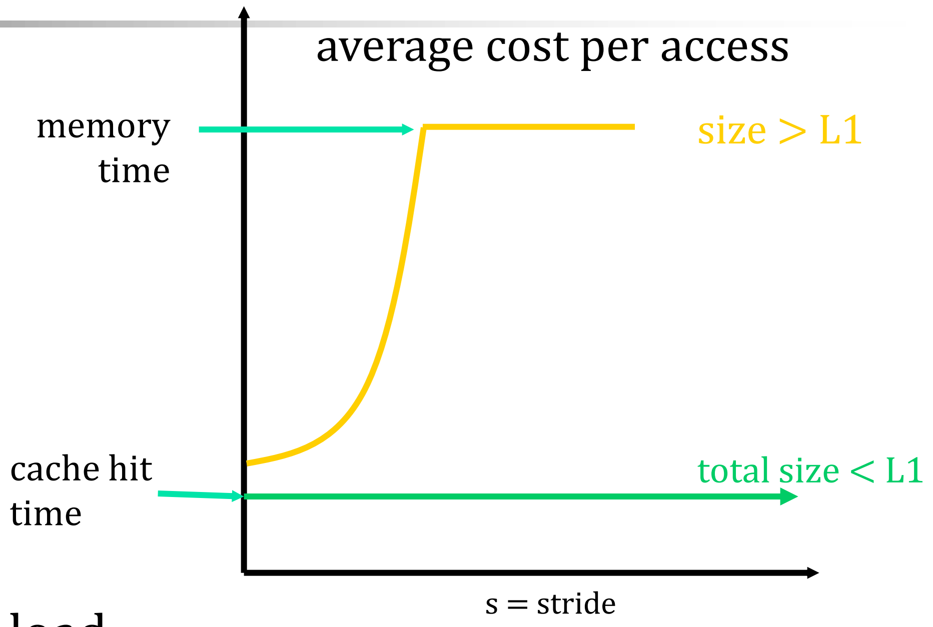
- Microbenchmark for memory system performance



```
•   for array A of length L from 4KB to 8MB by 2x
      for stride s from 4 Bytes (1 word) to L/2 by 2x
          time the following loop
          (repeat many times and average)
                  for i from 0 to L by s
                  load A[i] from memory (4 Bytes)
```

Membench: What to Expect





- Consider the average cost per load
  - Plot one line for each array length, time vs. stride
  - Small stride is best
  - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
  - Picture assumes only one level of cache
  - Values have gotten more difficult to measure on modern CPU

Outline

- Behind the compiler – the architecture basis
- **Tricks Compilers Play**
  - Scalar Optimizations
  - Loop Optimizations
  - Inlining
- Where is the problem and how to do
- Tricks You Can Play with Compilers
  - Introduction to Intel Compiler
  - Profiling

Compiler Design

The people who design compilers have a lot of experience working with the languages commonly used in High Performance Computing:

- Fortran: 50ish years
- C: 40ish years
- C++: 25ish years, plus C experience

So, they’ve come up with clever ways to make programs run faster.

Compiler Optimizations

- Copy Propagation
- Constant Folding
- Dead Code Removal
- Strength Reduction
- Common Subexpression Elimination
- Variable Renaming
- Loop Optimizations

Not every compiler does all of these, so it sometimes can be worth doing these by hand.

Much of this discussion is from [2] and [6].

Copy Propagation (C)

Before

```
x = y;
z = 1 + x;
```

Has data dependency

Compile

After

```
x = y;
z = 1 + y;
```

No data dependency

Constant Folding (C)

Before

```
add = 100;
aug = 200;
sum = add + aug;
```

After

```
sum = 300;
```

Notice that **`sum`** is actually the sum of two constants, so the compiler can precalculate it, eliminating the addition that otherwise would be performed at runtime.

Dead Code Removal (C)

Before

```
var = 5;
printf("%d", var);
exit(-1);
printf("%d", var * 2);
```

After

```
var = 5;
printf("%d", var);
exit(-1);
```

Since the last statement never executes, the compiler can eliminate it.

Strength Reduction (C)

Before

```
x = pow(y, 2.0);
a = c /  2.0;
```

After

```
x = y * y;
a = c * 0.5;
```

Raising one value to the power of another, or dividing, is more expensive than multiplying. If the compiler can tell that the power is a small integer, or that the denominator is a constant, it'll use multiplication instead.

Note: In C, "`pow(y, 2.0)`" means "y to the power 2."

What's the Relevance of Cycles?

Typically, a primitive operation (for example, add, multiply, divide) takes a fixed number of cycles to execute (assuming no pipelining).

- IBM POWER4

Multiply or add:	6 cycles (64 bit floating point)
Load:	4 cycles from L1 cache
	14 cycles from L2 cache

- Intel Pentium4 EM64T (Core)

Multiply:	7 cycles (64 bit floating point)
Add, subtract:	5 cycles (64 bit floating point)
Divide:	38 cycles (64 bit floating point)
Square root:	39 cycles (64 bit floating point)
Tangent:	240-300 cycles (64 bit floating point)

Common Subexpression Elimination (C)

Before

```
d = c * (a / b);
e = (a / b) * 2.0;
```

After

```
adivb = a / b;
d = c * adivb;
e = adivb * 2.0;
```

The subexpression `(a / b)` occurs in both assignment statements, so there's no point in calculating it twice.

This is typically only worth doing if **the common subexpression is expensive to calculate**.

Variable Renaming (C)

Before	After
`x = y * z;`	`x0 = y * z;`
`q = r + x * 2;`	`q = r + x0 * 2;`
`x = a + b;`	`x = a + b;`

The original code has an **output dependency**, while the new code **doesn’t** – but the final value of `x` is still correct.

Loop Optimizations

- Hoisting Loop Invariant Code
- Unswitching
- Iteration Peeling
- Index Set Splitting
- Loop Interchange
- Unrolling
- Loop Fusion
- Loop Fission

Not every compiler does all of these, so it sometimes can be worth doing some of these by hand.

Much of this discussion is from [3] and [6].

Hoisting Loop Invariant Code (C)

Code that doesn't change inside the loop is known as ***loop invariant***. It doesn't need to be calculated over and over.

Before

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + c * d;
  e = g(n);
}
```

After

```
temp = c * d;
for (i = 0; i < n; i++) {
  a[i] = b[i] + temp;
}
e = g(n);
```

Unswitching (C)

```
for (i = 0; i < n; i++) {
  for (j = 1; j < n; j++) {
    if (t[i] > 0)
      a[i][j] = a[i][j] * t[i] + b[j];
    }
    else {
      a[i][j] = 0.0;
    }
  }
}
```

The condition is `j`-independent.

Before

```
for (i = 0; i < n; i++) {
  if (t[i] > 0) {
    for (j = 1; j < n; j++) {
      a[i][j] = a[i][j] * t[i] + b[j];
    }
  }
  else {
    for (j = 1; j < n; j++) {
      a[i][j] = 0.0;
    }
  }
}
```

So, it can migrate outside the `j` loop.

After

Iteration Peeling (C)

Before

```
for (i = 0; i < n; i++) {
  if ((i == 0) || (i == (n - 1))) {
    x[i] = y[i];
  }
  else {
    x[i] = y[i + 1] + y[i - 1];
  }
}
```

We can eliminate the IF by ***peeling*** the weird iterations.

After

```
x[0] = y[0];
for (i = 1; i < n - 1; i++) {
  x[i] = y[i + 1] + y[i - 1];
END DO
x[n-1] = y[n-1];
```

Index Set Splitting (C)

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + c[i];
  if (i >= 10) {
    d[i] = a[i] + b[i - 10];
  }
}
```

Before

```
for (i = 0; i < 10; i++) {
  a[i] = b[i] + c[i];
}
for (i = 10; i < n; i++) {
  a[i] = b[i] + c[i];
  d[i] = a[i] + b[i - 10];
}
```

After

Note that this is a generalization of **peeling**.

Loop Interchange (C)

Before

```
for (j = 0; j < nj; j++) {
  for (i = 0; i < ni; i++) {
    a[i][j] = b[i][j];
  }
}
```

After

```
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    a[i][j] = b[i][j];
  }
}
```

Array elements **`a[i][j]`** and **`a[i][j+1]`** are near each other in memory, while **`a[i+1][j]`** may be far, so it makes sense to make the **`j`** loop be the inner loop. (This is reversed in Fortran.)

Unrolling (C)

Before

```
for (i = 0; i < n; i++) {
  a[i] = a[i] + b[i];
}
```

After

```
for (i = 0; i < n; i += 4) {
  a[i]   = a[i]   + b[i];
  a[i+1] = a[i+1] + b[i+1];
  a[i+2] = a[i+2] + b[i+2];
  a[i+3] = a[i+3] + b[i+3];
}
```

You generally **shouldn't** unroll by hand.

Why Do Compilers Unroll?

We saw last time that a loop with a lot of operations gets better performance (up to some point), especially if there are lots of arithmetic operations but few main memory loads and stores.

Unrolling creates multiple operations that typically load from the same, or adjacent, cache lines.

So, an unrolled loop has more operations without increasing the memory accesses by much.

Also, unrolling decreases the number of comparisons on the loop counter variable, and the number of branches to the top of the loop.

Loop Fusion (C)

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
}
for (i = 0; i < n; i++) {
  c[i] = a[i] / 2;
}
for (i = 0; i < n; i++) {
  d[i] = 1 / c[i];
}
```

Before

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] / 2;
  d[i] = 1 / c[i];
}
```

After

As with unrolling, this has fewer branches. It also has fewer total memory references.

Loop Fission (C)

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] / 2;
  d[i] = 1 / c[i];
}
```

Before

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
}
for (i = 0; i < n; i++) {
  c[i] = a[i] / 2;
}
for (i = 0; i < n; i++) {
  d[i] = 1 / c[i];
}
```

After

Fission reduces the cache footprint and the number of operations per iteration.

To Fuse or to Fizz?

The question of when to perform fusion versus when to perform fission, like many many optimization questions, is highly dependent on the application, the platform and a lot of other issues that get very, very complicated.

Compilers don’t always make the right choices.

That’s why it’s important to examine the actual behavior of the executable.

Inlining (C)

Before

```
for (i = 0;
     i < n; i++) {
  a[i] = func(i+1);
}
…
float func (x) {
  …
  return x * 3;
}
```

After

```
for (i = 0;
     i < n; i++) {
  a[i] = (i+1) * 3;
}
```

When a function or subroutine is ***inlined***, its contents are transferred directly into the calling routine, eliminating the overhead of making the call.

Outline

- Behind the compiler – the architecture basis
- Tricks Compilers Play
  - Scalar Optimizations
  - Loop Optimizations
  - Inlining
- **Where is the problem and how to do**
- Tricks You Can Play with Compilers
  - Introduction to Intel Compiler
  - Profiling

- Why is this your performance problem?

  In theory, compilers understand all of this and can optimize your program; in practice they don’t.

Metric for finding Poor single processor performance

- 机器的时钟周期为T_C，程序中指令总条数为I_N，执行每条指令所需的平均时钟周期数为**CPI**，则一个程序在CPU上运行的时间 T_{CPU}为：

$$T_{CPU} = I_N \times CPI \times T_C$$

MIPS(Million Instructions Per Second)

$$MIPS = I_N \ / \ (T_{CPU} \times 10^6) = R_C \ / \ (CPI \times 10^6)$$

MFLOPS(Million Floating Point Operations Per Second)

$$MFLOPS = I_{FN} \ / \ (T_{CPU} \times 10^6)$$

- 针对现代处理器，常用**IPC**来表示CPU部件利用率

如何增强串行程序性能

- **帮助**编译器
  - **隐藏内存访问开销**
  - **充分开发指令并行性**
- 使用高性能库开发程序
  - Intel MKL
  - AMD ACML
  - BLAS, LAPACK (www.netlib.org)
- Automatic optimization an active research area
  - BeBOP: bebop.cs.berkeley.edu/
  - PHiPAC: www.icsi.berkeley.edu/~bilmes/phipac

    in particular tr-98-035.ps.gz
  - **ATLAS: www.netlib.org/atlas**

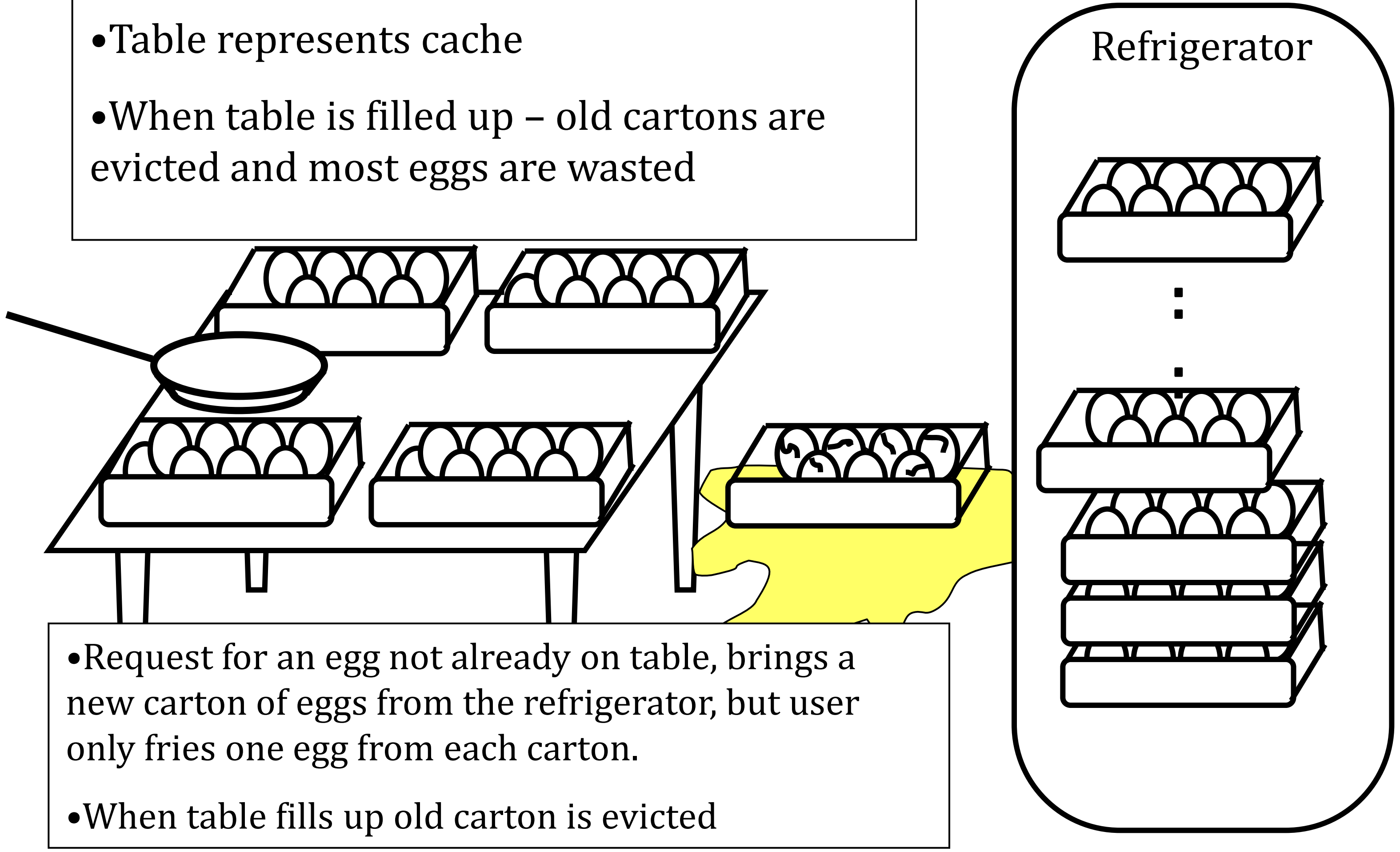
Approaches to Handling Memory Latency

- Bandwidth has improved more than latency
- Approach to address the memory latency problem

Poor Cache Utilizations - with Eggs

- Carton represents cache line
- Refrigerator represents main memory
- Table represents cache
- When table is filled up – old cartons are evicted and most eggs are wasted

User requests a 3rd egg – Carton evicted



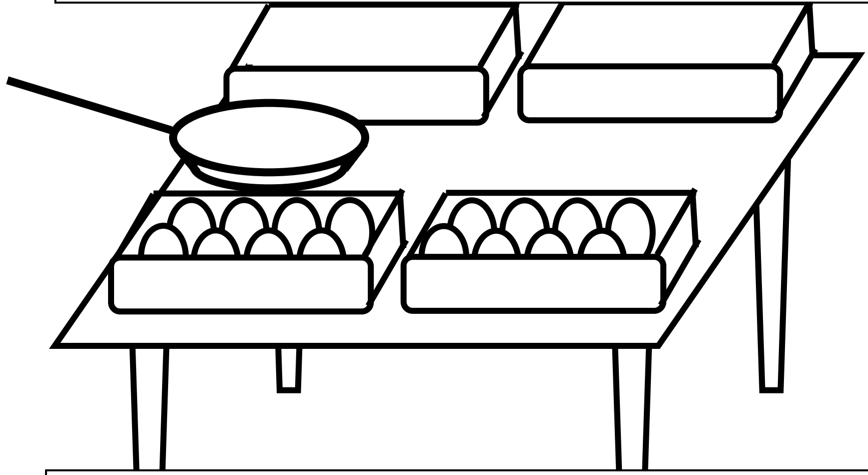
- Request for an egg not already on table, brings a new carton of eggs from the refrigerator, but user only fries one egg from each carton.
- When table fills up old carton is evicted

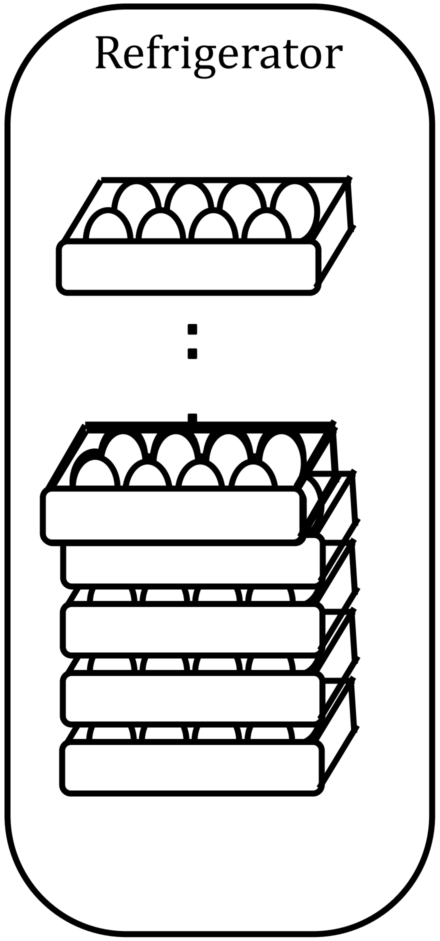
Good Cache Utilization - with Eggs

•Request for one egg brings new carton of eggs from refrigerator
User specifically requests eggs form carton already on table

•User fries all eggs in carton before egg from next carton is requested

User eventually asks for all the eggs





Carton eviction doesn’t hurt us because we’ve already fried all the eggs in the cartons on the table – just like previous user

Approaches to Handling Memory Latency

- Approach to address the memory latency problem
  - Eliminate memory operations by saving values in small, fast memory (cache) and reusing them
    - **need temporal locality in program**
  - Take advantage of better bandwidth by getting a chunk of memory and saving it in small fast memory (cache) and using whole chunk
    - **need spatial locality in program**
  - Take advantage of better bandwidth by allowing processor to issue multiple reads to the memory system at once
    - **concurrency in the instruction stream**

In Practise

- Alignment
  - Static memor: `__attribute__((aligned(n)))` in front variable declaration
  - Dynamic memory:

    `__mm_aligned_malloc(size, alignment_bytes);__mm_aligned_free()`
  - Using Intel® Threading Building Blocks

    `scalable_aligned_malloc()/scalable_aligned_free()`
- Cache blocking
- Prefetch
- Loop fusion and Loop distribution
- AOS->SOA

```
struct Point{   //AoS
float r;
float r;
float g;
}
```

```
struct Points{ //SoA
float* x;
float* y;
float* z;
}
```

R	G	B	R	G	B	R	G	B

R	R	R	G	G	G	B	B	B

Lessons

- Actual performance of a simple program can be a complicated function of the architecture
  - Slight changes in the architecture or program change the performance significantly
  - To write fast programs, need to consider architecture
    - True on sequential or parallel processor
  - We would like simple models to help us design efficient algorithms

- We will illustrate with a common technique for improving cache performance, called blocking or tiling
  - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache

Why Matrix Multiplication?

- An important kernel in scientific problems
  - Appears in many linear algebra algorithms
  - Closely related to other algorithms
- Optimization ideas can be used in other problems
- The best case for optimization payoffs
- The most-studied algorithm in high performance computing
- May be one part of your project

Note on Matrix Storage

- A matrix is a 2-D array of elements, but memory addresses are “1-D”
- Conventions for matrix layout
  - by column, or “column major” (Fortran default); A(i,j) at A+i+j*n
  - by row, or “row major” (C default) A(i,j) at A+i*n+j

Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

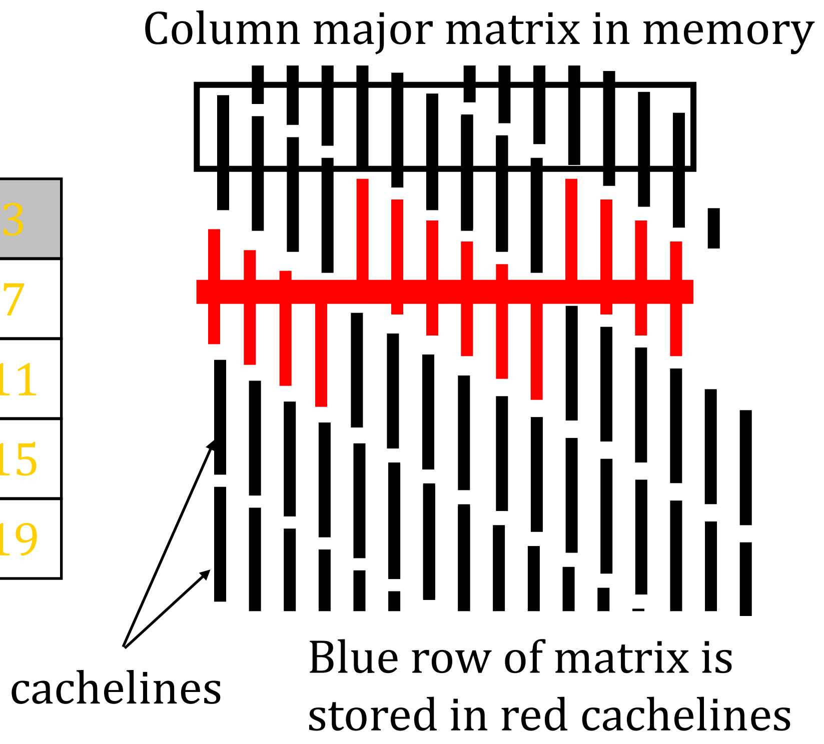

Column major matrix in memory
cachelines
Blue row of matrix is stored in red cachelines

Figure source: Larry Carter, UCSD

- Column major (for now)

Using a Simple Model of Memory to Optimize

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation $<< t_m$
  - q = f / m average number of flops per slow memory access

Computational Intensity: Key to algorithm efficiency

- Minimum possible time = f* t_f when all data in fast memory
- Actual time
  - $f * t_f + m * t_m = f * t_f * (1 + \boxed{t_m/t_f} * 1/q)$

*Machine Balance:*Key to machine efficiency

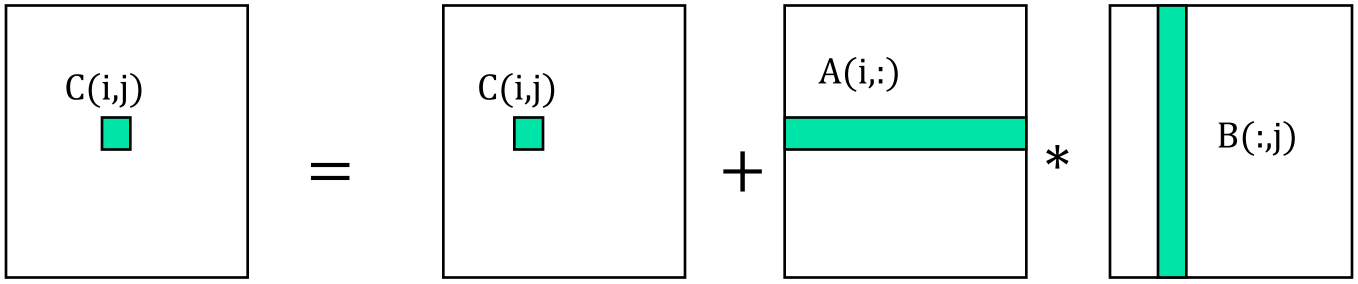
- Larger q means time closer to minimum f * t_f
  - $q \geq t_m/t_f$ needed to get at least half of peak speed

Naive Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

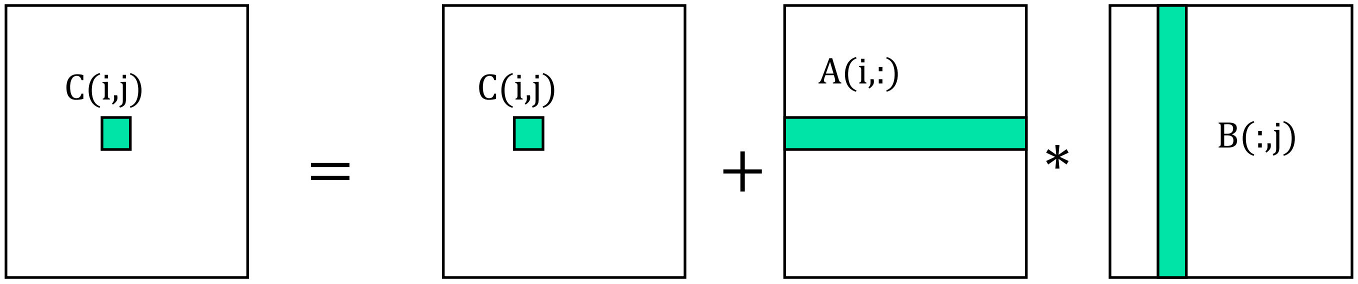
Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory

q potentially as large as $2*n^3 / 3*n^2 = O(n)$



Naive Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
 {read row i of A into fast memory}
  for j = 1 to n
    {read C(i,j) into fast memory}
    {read column j of B into fast memory}
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    {write C(i,j) back to slow memory}
```



Naive Matrix Multiply

Number of slow memory references on unblocked matrix multiply

$m = n^3$ read each column of B n times

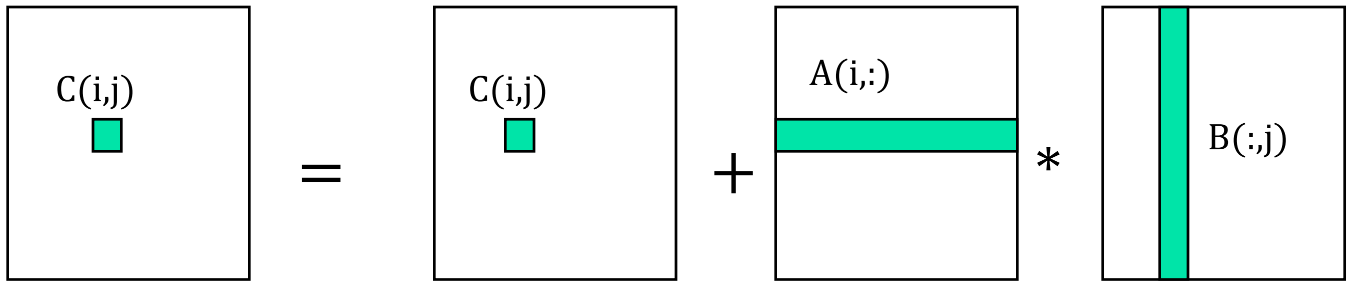
$+ n^2$ read each row of A once

$+ 2n^2$ read and write each element of C once

$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (n^3 + 3n^2)$

$\sim= 2$ for large n

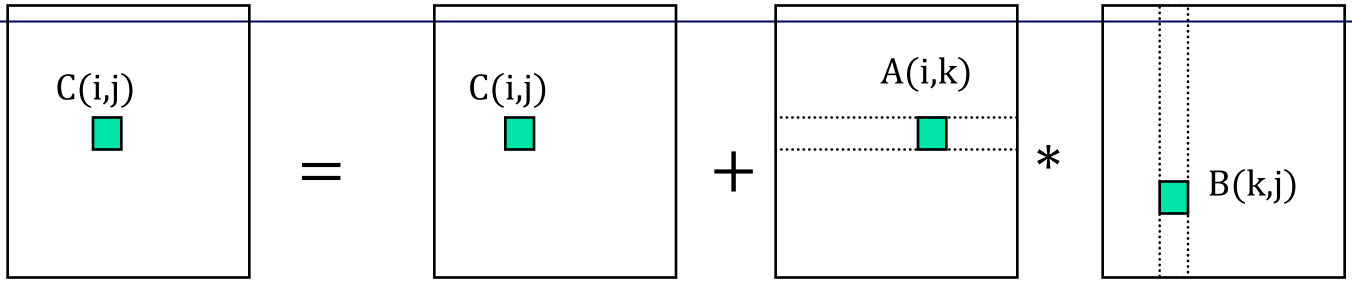


Blocked Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

b=n / N is called the block size

```
for i = 1 to N
  for j = 1 to N
    {read block C(i,j) into fast memory}
    for k = 1 to N
      {read block A(i,k) into fast memory}
      {read block B(k,j) into fast memory}
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
  {write block C(i,j) back to slow memory}
```



Blocked (Tiled) Matrix Multiply

Recall:

m is amount memory traffic between slow and fast memory

matrix has nxn elements, and NxN blocks each of size bxb

f is number of floating point operations, $2n^3$ for this problem

$q = f / m$ is our measure of algorithm efficiency in the memory system

So:

$m = N*n^2$ read each block of B N^3 times ($N^3 * b^2 = N^3 * (n/N)^2 = N*n^2$)

$+ N*n^2$ read each block of A N^3 times

$+ 2n^2$ read and write each block of C once

$= (2N + 2) * n^2$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$\sim= n / N = b$ for large n

So we can improve performance by increasing the blocksize b

Can be much faster than naive method (q=2)

Using Analysis to Understand Machines

The blocked algorithm has computational intensity q ~= b

- The larger the block size, the more efficient our algorithm will be
- Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large
- Assume your fast memory has size M_{fast}

$$3b^2 <= M_{fast}, \text{ so } q \sim= b <= \text{sqrt}(M_{fast}/3)$$

- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size

$$M_{fast} >= 3b^2 \sim= 3q^2 = 3(t_m/t_f)^2$$

- This size is reasonable for L1 cache, but not for register sets
- Note: analysis assumes it is possible to schedule the instructions perfectly

	required KB	
Ultra 2i	14.8	
Ultra 3	4.7	
Pentium 3	0.9	
Pentium3M	2.4	
Power3	1.8	
Power4	5.4	
Itanium1	31.1	
Itanium2	0.7	

Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving)
  - www.netlib.org/blas,   www.netlib.org/blas/blast--forum
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy (y=a*x+y), etc
    - m=2*n, f=2*n, q ~1 or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - m=n^2, f=2*n^2, q~2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - m <= 3n^2, f=O(n^3), so q=f/m can possibly be as large as n, so BLAS3 is potentially much faster than BLAS2
- Good algorithms used BLAS3 when possible (LAPACK & ScaLAPACK)
  - See www.netlib.org/{lapack,scalapack}
  - More later in course

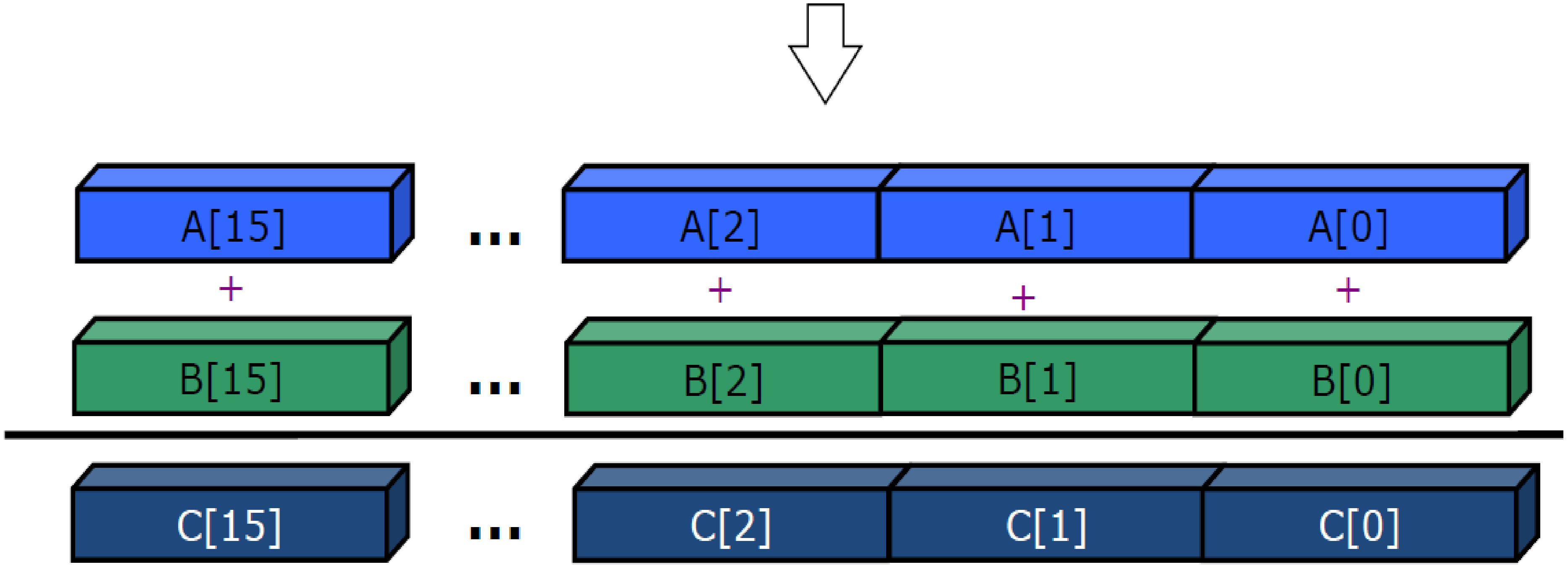
Summary

- Performance programming on uniprocessors requires
  - understanding of memory system
  - understanding of fine-grained parallelism in processor
- Simple performance models can aid in understanding
  - Two ratios are key to efficiency (relative to peak)
  1. computational intensity of the algorithm:
     - $q = f/m$ = # floating point operations / # slow memory references
  2. machine balance in the memory system:
     - t_m/t_f = time for slow memory reference / time for floating point operation
- Want $q > t_m/t_f$ to get half machine peak
- Blocking (tiling) is a basic approach to increase q
  - Techniques apply generally, but the details (e.g., block size) are architecture dependent
  - Similar techniques are possible on other data structures and algorithms

What is SIMD- Translate Loops into SIMD Parallelism

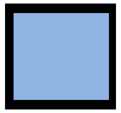
Vector (or SIMD) Code computes more than one element at a time. SIMD stands for Single Instruction Multiple Data.

```
for (i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```



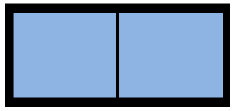
History of SIMD ISA extensions

Intel® Pentium® processor (1993)

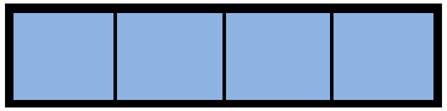


MMX™ (1997)

Illustrated with the number of 32bit data processed by one "packed" instruction

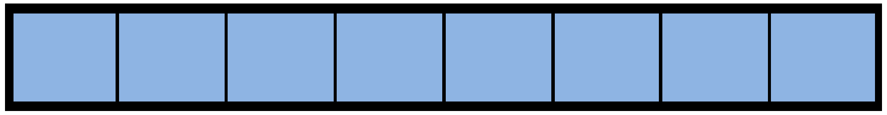


Intel® Streaming SIMD Extensions (Intel® SSE in 1999 to Intel® SSE4.2 in 2008)



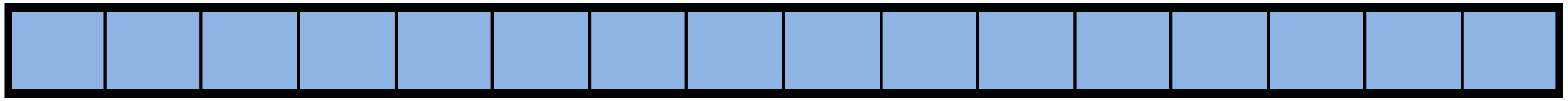
128bit- 4 SP elements one time

Intel® Advanced Vector Extensions (Intel® AVX in 2011 and Intel® AVX2 in 2013)



256bit- 8 SP elements one time

Intel Many Integrated Core Architecture (Intel® MIC Architecture in 2012)



512 bit- 16 SP elements one time

Vectorization

- Auto-vectorization
- SIMD pragma
- CEAN
- Elemental function
- Vector Intrinsic and Vector Classes
- Inline ASM

本课程我们主要关注采用编译器实现向量化，具体内容将在下节介绍

Thanks for your attention!

References

[1] Kevin Dowd and Charles Severance, *High Performance Computing,* 2nd ed. O'Reilly, 1998, p. 173-191.

[2] Ibid, p. 91-99.

[3] Ibid, p. 146-157.

[4] NAG **f95** man page, version 5.1.

[5] Intel **ifort** man page, version 10.1.

[6] Michael Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Co., 1996.

[7] Kevin R. Wadleigh and Isom L. Crawford, *Software Optimization for High Performance Computing*, Prentice Hall PTR, 2000, pp. 14-15.