Scientific and Technical Computing

Hardware and Code Optimization

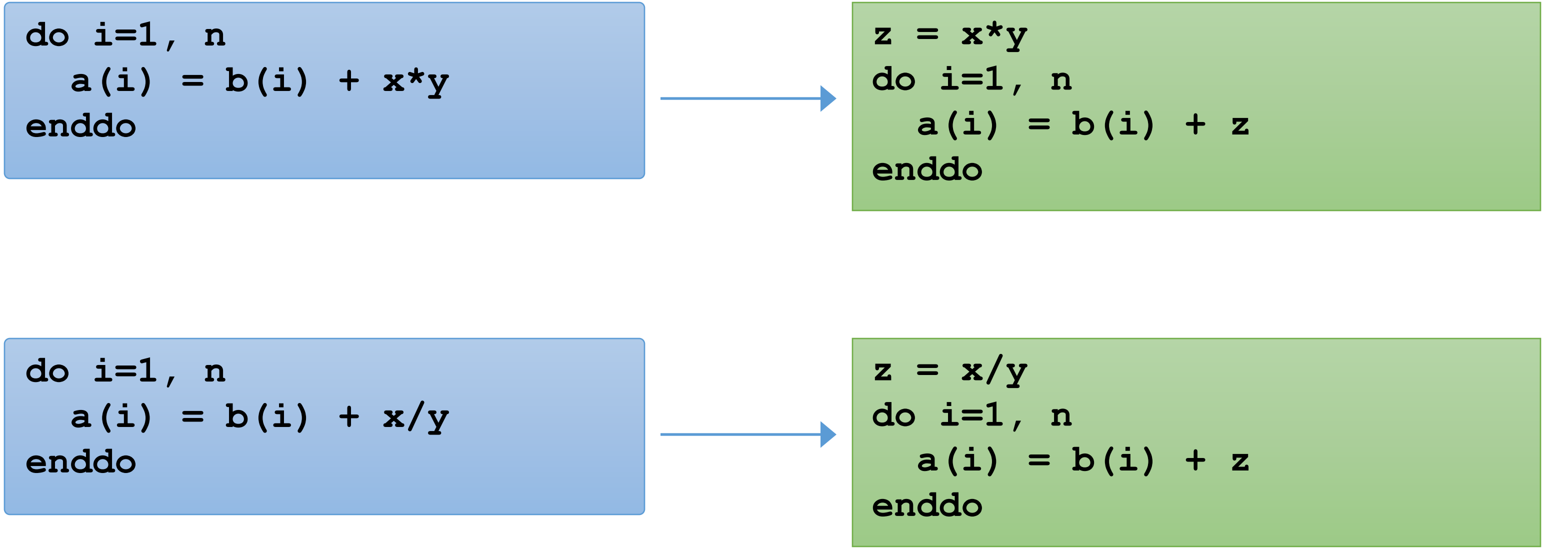
Lars Koesterke

UT Austin, 10/19/21 & 11/4/21

Optimizations in Code

Optimizations in Code

Moving invariant code outside the loop



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

```
z = x*y
do i=1, n
  a(i) = b(i) + z
enddo
```

```
do i=1, n
  a(i) = b(i) + x/y
enddo
```

```
z = x/y
do i=1, n
  a(i) = b(i) + z
enddo
```

Optimizations in Code

Cost of a division: maybe 60 cycles (depends highly on precision and accuracy)

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

→

```
xi = 1. / x
do i=1, n
  a(i) = b(i) + c(i) * xi
enddo
```

Note that the following is not exactly (bit-wise) the same

```
z1 = y / x

z2 = y * (1./x)
```

Do no assume that z1 is exactly/bit-wise z2

If you change the code, or if you allow the compiler to change the code,
you will get a slightly different result (last bits will vary)
Compiler options allow you to control (to some degree)

- Whether operations in code are replaced by cheaper operations
- Whether operations are 'executed' to the fullest accuracy (which is expensive)
  - The default is some 'reasonable' compromise between accuracy and speed

Many optimization techniques will change the rounding bits.

Hence, bit-wise reproducibility is not the goal

Optimizations in Code

Changing result by reordering operations

Assume that the data representation has **4 significant digits**

```
x = 1.e-5
s = 0.
do i=1, 10000
  s = s + x
enddo
s = s + 1.

print s
```

```
x = 1.e-5
s = 1.
do i=1, 10000
  s = s + x
enddo

print s
```

Do we agree that the 2 code versions are intended to calculate the same sum?

Optimizations in Code

Changing result by reordering operations

Assume that the data representation has 4 significant digits

```
x = 1.e-5
s = 0.
do i=1, 10000
  s = s + x
enddo
s = s + 1.
print s
```

→

```
x = 1.e-5
s = 1.
do i=1, 10000
  s = s + x
enddo

print s
```

What is being printed?

What is 1. + 1.e-5?

Optimizations in Code

Changing result by reordering operations

Assume that the data representation has 4 significant digits

```
x = 1.e-5
s = 0.
do i=1, 10000
   s = s + x
enddo
s = s + 1.
print s              1.1
```

```
x = 1.e-5
s = 1.
do i=1, 10000
   s = s + x
enddo

print s                         1.
```

Do not hope for bit-wise reproducible results

Too many things out of your control will change actual numeric

Aiming for bit-wise reproducibility will prevent you from

Achieving high performance

Reproducible results are a key ingredient of the ‘scientific method’

But bit-wise reproducibility is not required

Optimizations in Code

What is the problem here (in terms of code performance)?
Independent of what actual language is being used (C or Fortran)

```
do j=1,n
  do i=1,n
    a(i,j)=a(i,j)+s*b(j,i)
  end do
end do
```

Optimizations in Code

What is the problem here (in terms of code performance)?

```
do j=1,n
  do i=1,n
    a(i,j)=a(i,j)+s*b(j,i)
  end do
end do
```

One array is accessed stride-1

One array is accessed with a high stride

Changing the loop order does not help

Optimizations in Code

Strip mining :

Transforms a single loop into two loops to insure that each element of a cache line is used, once it is brought into the cache. In the example below, the loop over j is split into two:

```
do j=1,n
  do i=1,n
    a(i,j)=a(i,j)+s*b(j,i)
  end do
end do
```

→

```
do jouter=1,n,8
  do j=jouter,min(jouter+7,n)
    do i=1,n
      a(i,j)=a(i,j)+s*b(j,i)
    end do
  end do
end do
```

↓

```
do jouter=1,n,8
  do i=1,n
    do j=jouter,min(jouter+7,n)
      a(i,j)=a(i,j)+s*b(j,i)
    end do
  end do
end do
```

Memory Tuning

Array Blocking

The objective of array blocking is to work with small array blocks when expressions contain mixed-stride operations. It uses complete cache lines when they are brought in from memory, and hence avoid possible eviction that would otherwise ensue without blocking.

```
do i=1,n
do j=1,n
 A(j,i)=B(i,j)
end do
end do
```

```
do i=1,n,2   (increment of 2)
do j=1,n,2
 A(j  ,i  )=B(i  ,j  )
 A(j+1,i  )=B(i+1,j  )
 A(j  ,i+1)=B(i  ,j+1)
 A(j+1,i+1)=B(i+1,j+1)
end do
end do
```

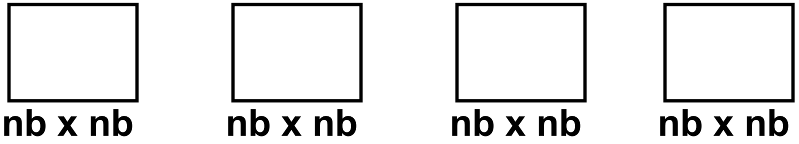
Memory Tuning

Array Blocking for matrix x matrix multiplication

```
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb
   do jj=1,n,nb
      do kk=1,n,nb
         do i=ii,min(n,ii+nb-1)
            do j=jj,min(n,jj+nb-1)
               do k=kk,min(n,kk+nb-1)

            c(i,j)=c(i,j)+a(j,k)*b(k,i)





end do; end do; end do; end do; end do; end do
```

Simple implementation
has 3 loops

Let’s not get lost in the details here



Cache Set Associativity

Caches are divided into “sets”.

Set associativity is the number of cache lines that can be stored within each set.

– Direct Mapped = 1-way set associative

– k-way set associative (2**n; n=1,2,3…)

– Fully associative

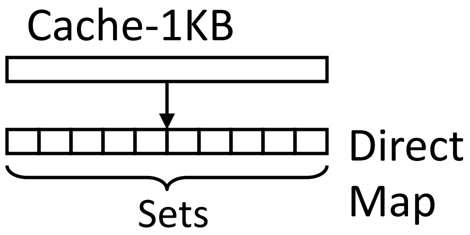
Association refers to: the correct data of a set is returned by associating address.

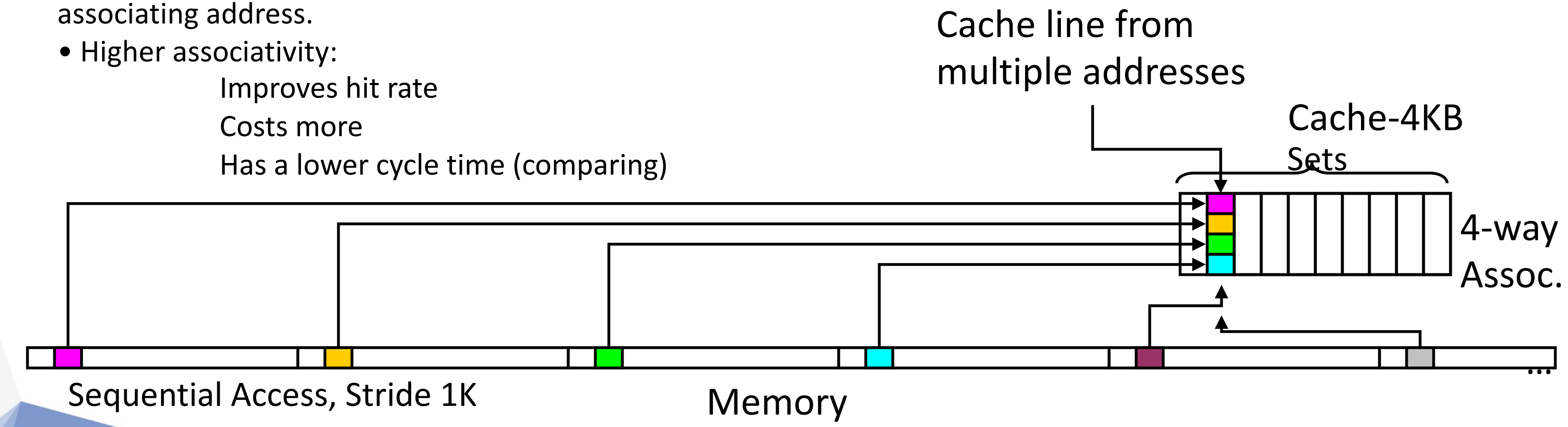
- Higher associativity:

  Improves hit rate

  Costs more

  Has a lower cycle time (comparing)




Cache line from multiple addresses
Cache-4KB
Sets
4-way Assoc.
...
Sequential Access, Stride 1K
Memory

Cache Tuning

Memory access with a stride of a high power of two usually leads to this form of cache thrashing, because data cache sizes are a (high) power of two in bytes. The following **cache trashing** code references array elements with a stride of 8KB:

```
program MAIN
integer, parameter :: n=1000
real*8 :: A(1024,n), B(1024,n)
real*8 :: C(1024,n), D(1024,n)
common /Arrays/ A,B,C,D
...
do i=1,n
   D(64,i)= c1*A(64,i)+c2*B(64,i)+c3*C(64,i)
end do
...
end program
```

```
Integer, Parameter :: n=1000, pad=24
Real*8 :: A(1024+pad,n)
Real*8 :: B(1024+pad,n)
Real*8 :: C(1024+pad,n)
Real*8 :: D(1024+pad,n)
```

More is better!

But then there are also power concerns …

How to sell a new computer

to somebody who already has a computer (or 2, or 3, … or 1000)

It is all about one number

Stampede2 Skylake node: Theoretical maximum floating point performance

2 *	2 *	8 *	48 *	2.1 GHz	~ 2200 GFlops
2 VPUs	FMA	8 Vector Lanes	Cores	Frequency (nominal)	Flops = floating point operations per sec

FMA: Fused Multiply-Add

Frequency: nominally 2.1GHz (+ Turbo modes) → fuzzy math

Raw performance is *the* characteristic number used to rate performance

Typical application performance is vastly lower, though (maybe, on average, 1% of peak)

Bottleneck: Memory Bandwidth

2 *   2 * 8 *   48 * 2.1 GHz ~ 2200 Gflops

Application performance is typically 1% of peak because:

- Memory bandwidth insufficient (data transfer from main memory to registers)
- Hardware can execute 100 Flops per word loaded (stored) from (to) memory
- Inefficiencies in code and/or algorithm regarding
  - Memory access
  - Vectorization
  - Pipelining
  - Re-use of data with caches
  - Potentially/likely insufficient parallelism (on all levels)
  - …

Hardware Design Implications

2 *   2 * 8 *   48 * 2.1 GHz ~ 2200 Gflops

How to achieve peak performance with limited memory bandwidth?

→ Clever benchmark: Matrix-Matrix-Multiply

Number of operations: ~n^3

Number of data elements: ~n^2

→ With data re-use (temporary storage in caches): bandwidth / flops drops with ~ 1/n (theoretical)

→ Also note how the FMA concept fits the dot-product in the benchmark perfectly

Benchmark performance with caches: 90%+ of peak

Sidenote

Two ‘Crazy’ Ideas (mainly from the CS Department)

Why not use hardware easier to use?

- Less peak, but more useable performance

Why not develop a compiler that does all the heavy lifting for you?

- Rearrange the code on a large scale (not just inner loops)
- Improve memory access pattern
- Add parallelism

Hardware Design Implications

2 *	2 *	8 *	48 *	2.1 GHz	~ 2200 Gflops
2 VPUs	FMA	8 Vector Lanes	Cores	Frequency (nominal)	Flops = floating point operations per sec

Very high peak floating point performance through concurrency/parallelism in hardware

- vector processing units (2x), fused multiply-add (2x), vectorization (8x)
- pipelining

Complex memory architecture

- Multiple levels of caches on chip (L1 & L2 for each core, L3 for each socket/CPU)
    - Caches use a different mechanism for storage compared to DDR memory
- Data transfer in bulk (cache lines) & prefetching

Benchmark code has all the bells and whistles to exploit all features (your code does not)

Saving Power with Multiple Cores

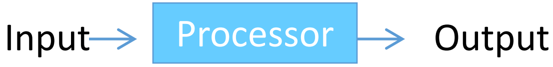
Power = CV^2F

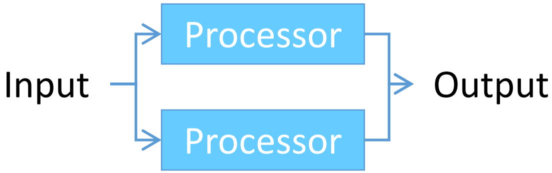
Input → Processor → Output

Cap = 1.0c
Volts = 1.0v
Freq = 1.0f
Power = $1.0cv^2f$

Saving Power with Multiple Cores

Power = CV^2F





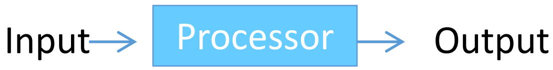
Cap = 1.0c
Volts = 1.0v
Freq = 1.0f
Power = $1.0cv^2f$

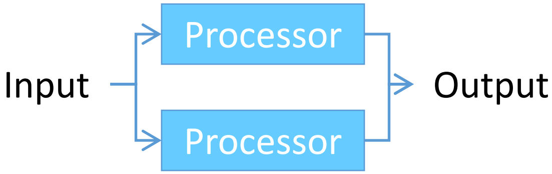
Do same work with 2 processors at 0.5 freq.

Voltage ~ Frequency
2x more wires ->
~2x Capacitance

Saving Power with Multiple Cores

Power = CV^2F





Cap = 1.0c
Volts = 1.0v
Freq = 1.0f
Power = $1.0cv^2f$

Do same work with 2 processors at 0.5 freq.

Voltage ~ Frequency
2x more wires ->
~2x Capacitance

Cap = 2.2c
Volts = 0.6v
Freq = 0.5f
Power = $0.4cv^2f$

about 40% of the original consumption

Hardware Design Implications

Concepts we have discussed (some in great detail)

- Turbo mode: Frequency changes with load (2.1GHz; full load: 1.4 GHz; single core 3.0 GHz)
- VPU, FMA: concurrent execution of 2 pairs of multiply-add (super-scalar architecture)
- Data streams
- Pipelining: 1 result per clock cycle (tick), although an operation (add/mult) takes 3-5 cycles
  - Think ‘assembly line’
- Vectorization: concurrent execution of 8 floating point operations (16 in single prec.)
- Caches & cache hierarchies: fast memory close to registers
  - Also **c** (speed of light) is limited: 1 tick @ 2GHz translates to 15 cm
- Cache lines: memory transfer in bulk
  - Data moves in chunks of 512 bit (8 double or 16 single)
- Prefetching: memory transactions anticipated through pattern recognition

Why do you have to use all cores?

Turn on **the light** at home

The current flows from the power plant through the switch and through the light bulb

How fast is the electrical current?   (The light turns on instantly)

Why do you have to use all cores?

Turn on the **water faucet** at home

The water flows from the water treatment plant through the pipes

How fast is the water flowing?   (The water flows instantly)

Why do you have to use all cores?

Turn on the **water faucet** at home

The water flows from the plant through the pipes

How fast is the water flowing? **(The water flows instantly)**

The light bulb turns on instantly and the water flows instantly(*) because the ‘pipe’ is already filled with electron and water, respectively.

Same is through for our data pipeline, as we have discussed before.

But how much data do we need to fill the data pipeline?

(*)Instantly means: according to the speed of light and speed of sound

Why do you have to use all cores?

Some numbers on bandwidth and latency (Skylake, per socket)

Bandwidth = 120 GB/s → 2 billion cache lines per second

Latency = 70 ns → 140 cache lines in the data stream

Therefore at least 140 outstanding memory requests are needed

to fill the pipeline

Why do you have to use all cores?

Some numbers on bandwidth and latency (Skylake, per socket in units per second)

Bandwidth = 120 GB/s → 2 billion cache lines per second

Latency = 70 ns → 140 cache lines in the data stream

Therefore at least 140 outstanding memory requests are needed

to fill the pipeline

Each core can issue about 10 memory requests before the core's 'book keeping' capability is exhausted.

Capability: tracking a limited number of unsatisfied/outstanding requests

Why do you have to use all cores?

140 requests in pipeline vs. 0 requests per core

Implications:

Per socket at least 14 cores (out of the 24) must be used to achieve peak memory bandwidth

Why not ‘more requests per socket’?

The designers assume that applications will use all cores

Over engineering the request table (per core) costs valuable space on the die

and will negatively impact other features

Next semester: PCSE

Parallel **C**omputing for **S**cience and **E**ngineering

Parallel computing with OpenMP and MPI

(hope to see some of you there)

Refresher

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

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

RAW

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

WAR

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

WAW

Do you get the same result when running forward and backward?

Can these loops be vectorized?

Definition of ‘WAR’, ‘RAW’, and ‘WAW’ at: cvw.cac.cornell.edu/vector/coding_dependencies

Adding numbers: concurrent and serial components (unsigned integers)

```
11 + 7 = ?
```

Adding numbers: concurrent and serial components

```
11 + 7 = ?
```

```
01011
00111
-----
```

<u>Concurrency</u>

<u>Serial</u>

Adding numbers: concurrent and serial components

```
11 + 7 = ?
```

```
01011
00111
-----
01122     ‘intermediate’

10010     ‘carry-over’ right to left
```

Concurrency

Adding-up all the digits

Serial

Moving the carry over from right to left

Multiplying numbers: concurrent and serial components (unsigned integers)

```
11 * 7 = ?
```

```
     01011 x 00111
------------------
00000
 00000
   01011
    01011
     01011
----------
0000112221    ‘intermediate’

0001001101    ‘carry-over’ right to left
```

Concurrency

Adding-up all the digits

Serial

Moving the carry over from right to left

For floating point numbers this is more complicated

But in principle it is understandable why only few cycles (3-5) are needed

There are many(!) aspects that we have not covered

But we have discussed all the main hardware components and optimization techniques