AMath 483/583 — Lecture 11

Outline:

- Computer architecture
- Cache considerations
- Fortran optimization

Reading:

- S. Goedecker and A. Hoisie, “Performance Optimization of Numerically Intensive Codes”, SIAM, 2001. ebook edition: `http://epubs.siam.org/doi/book/10.1137/1.9780898718218`

  UW students can also access from off-campus using the Library Proxy

- class notes: Bibliography on Other Courses...

Comments on Homework

Homework 2 should be graded soon.

For Homework 3 and future homeworks, SHA-1 hash must be submitted to web form by 11:00 pm on Wednesday for full credit. Don't put off until last minute!

My office hours today will be 9:30–9:55 am.

AMath 483/583 — Lecture 11

Outline:

- Computer architecture
- Cache considerations
- Fortran optimization

Reading:

- S. Goedecker and A. Hoisie, “Performance Optimization of Numerically Intensive Codes”, SIAM, 2001. ebook edition: `http://epubs.siam.org/doi/book/10.1137/1.9780898718218`

  UW students can also access from off-campus using the Library Proxy

- class notes: Bibliography on Other Courses...

How fast are computers?

Kilo = thousand (10^3)
Mega = million (10^6)
Giga = billion (10^9)
Tera = trillion (10^{12})
Peta = 10^{15}
Exa = 10^{18}

Processor speeds usually measured in Gigahertz these days.

Hertz means "machine cycles per second".

One operation may take a few cycles.

So a 1 GHz processor (10^9 cycles per second) can do
$> 100,000,000$ floating point operations per second
(> 100 Megaflops).

The Cray-1 computer

- World's first "supercomputer"
- Sold to Los Alamos, NCAR, etc. starting in 1976
- Price: up to $8.8 million

The Cray-1 computer

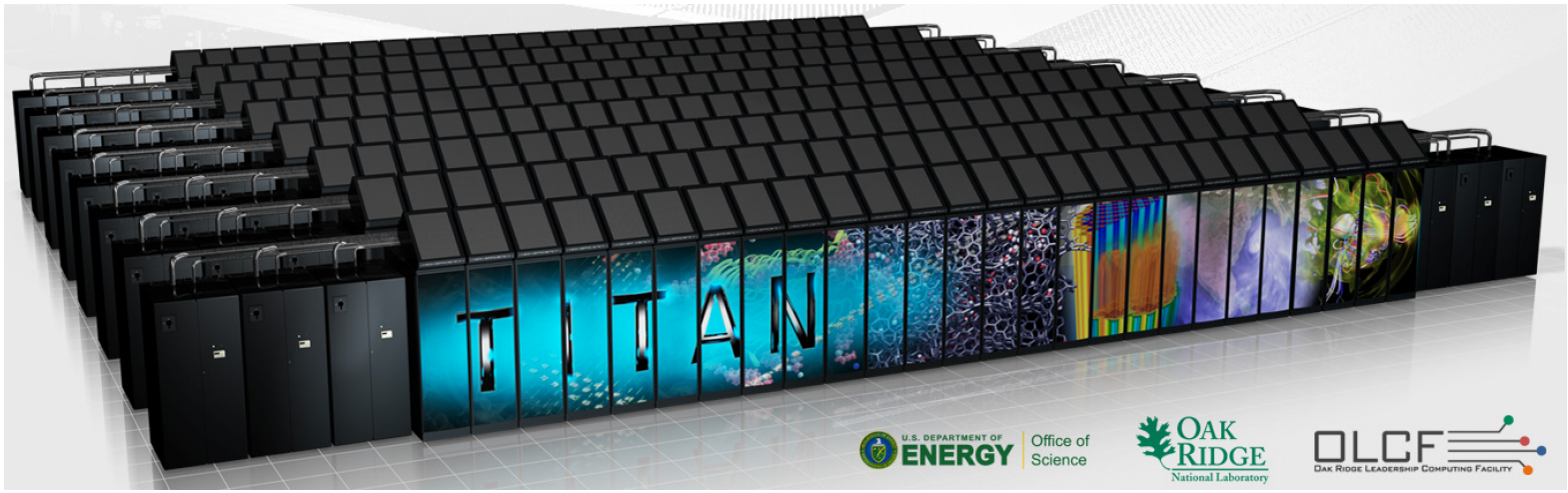
- World's first "supercomputer"
- Sold to Los Alamos, NCAR, etc. starting in 1976
- Price: up to $8.8 million
- Speed: 80-100 Mflops
- Memory: 8MB

Overview

High Performance Computing (HPC) often means heavy-duty computing on clusters or supercomputers with 100s of thousands of cores.

"World's fastest computer"

#1. Titan (Oak Ridge National Lab): 560,640 cores,
≈ 20 Petaflops $= 20,000,000,000,000,000$ flops



See `http://top500.org` for current list.

How fast are computers?

Not long ago counting flops was the best way to measure performance for scientific computing.

How fast are computers?

Not long ago counting flops was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.

If A and B are $n \times n$ then so is C.

Each element c_{ij} is the inner product of
ith row of A with jth column of B.

Requires n multiplications and $n - 1$ additions to compute c_{ij}.

How fast are computers?

Not long ago counting flops was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.

If A and B are $n \times n$ then so is C.

Each element c_{ij} is the inner product of
ith row of A with jth column of B.

Requires n multiplications and $n - 1$ additions to compute c_{ij}.

n^2 elements in $C \implies$ Requires $\mathcal{O}(n^3)$ floating point ops total.
Note: $n = 10,000 \implies n^3 = 10^{12}$
($> 1,000$ seconds on 1 GHz processor)

How fast are computers?

Not long ago counting flops was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.

If A and B are $n \times n$ then so is C.

Each element c_{ij} is the inner product of
ith row of A with jth column of B.

Requires n multiplications and $n - 1$ additions to compute c_{ij}.

n^2 elements in $C \implies$ Requires $\mathcal{O}(n^3)$ floating point ops total.
Note: $n = 10,000 \implies n^3 = 10^{12}$
($> 1,000$ seconds on 1 GHz processor)

But these days, the bottle neck is often
getting data to and from the processor!

Note that each element of A, B is used n times.

Memory Hierachy

(Main) Memory: "Fast" memory that is hopefully large enough to contain all the programs and data currently running.

(But not nearly fast enough to keep up with CPU.)

Typically 1 – 4 GB.

Recall GB = gigabyte = 10^9 bytes = 8×10^9 bits.

For example, 1GB holds a single $10,000 \times 10,000$ matrix of floating point values (8 bytes each),
or 125 matrices that are each 1000×1000.

Hard Drive: Slower memory that contains data (including photos, video, music, etc.) and all programs you might want to use.

Typically 80 – 500 GB. (Slower but cheaper.)

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer.
On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion distinct addresses}$$

$\Longrightarrow$ at most 4GB of memory can be addressed.

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer.
On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion distinct addresses}$$

$\Longrightarrow$ at most 4GB of memory can be addressed.

Newer machines often have more, leading to the need for 64-bit architectures (8 bytes for addresses).

$$2^{64} = 1.84 \times 10^{19} \text{ distinct addresses}$$

$\Longrightarrow$ could address an exabyte of memory.

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer.
On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion distinct addresses}$$

$\Longrightarrow$ at most 4GB of memory can be addressed.

Newer machines often have more, leading to the need for 64-bit architectures (8 bytes for addresses).

$$2^{64} = 1.84 \times 10^{19} \text{ distinct addresses}$$

$\Longrightarrow$ could address an exabyte of memory.

Note: Integers might still be stored in 4 bytes, for example.

Floats might be either `real(kind=4)` or `real(kind=8)`.

CPU and registers

CPU — central processor unit

Executes instructions such as add or multiply.

Takes data from registers, performs operations, stores back to registers.

Transferring between registers and processor is very fast.

Different types of registers, e.g.

- Integer, floating point
- instruction registers
- address registers

Generally a very small number of registers.

Data and instructions must be transferred between other memory and registers as needed.

Memory Hierachy

Between registers and memory there are 2 or 3 levels of cache, each larger but slower.

Registers: access time 1 cycle

L1 cache: a few cycles

L2 cache: $\sim$ 10 cycles

(Main) Memory: $\sim$ 250 cycles

Hard drive: 1000s of cycles

Terminology

Latency refers to amount of time it takes to complete a given unit of work.

Throughput refers to the amount of work that can be completed per unit time.

Terminology

Latency refers to amount of time it takes to complete a given unit of work.

Throughput refers to the amount of work that can be completed per unit time.

Exploit parallelism to hide latency and increase throughput.

Even a "single core" machine has lots of things going on at once.

For example:

- Pipelined operations
- Executing / fetching / storing
- Prefetching future instructions
- Prefetching data into cache

5-stage instruction pipeline for RISC machine

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**

IF = Instruction Fetch,
ID = Instruction Decode,
EX = Execute,
MEM = Memory access,
WB = Register write back.

http://en.wikipedia.org/wiki/Instruction_pipeline

Reducing memory latency

Reduce memory fetches by reusing data in cache as much as possible. Requires temporal locality.

Very simple Python example: if `len(x)` much larger than cache size,

```
z = 0.; w = 0.
for i in range(len(x)):
    z = z + x[i]
for i in range(len(x)):
    w = w + 3. * x[i]
```

should be rewritten as

```
for i in range(len(x)):
    z = z + x[i]
    w = w + 3. * x[i]
```

Reducing memory latency

Reduce memory fetches by reusing data in cache as much as possible. Requires temporal locality.

Very simple Python example: if `len(x)` much larger than cache size,

```
z = 0.; w = 0.
for i in range(len(x)):
    z = z + x[i]
for i in range(len(x)):
    w = w + 3. * x[i]
```

should be rewritten as

```
for i in range(len(x)):
    z = z + x[i]
    w = w + 3. * x[i]
```

Note: Both are bad in Python, use e.g. `z = np.sum(x); w = 3*z`

Cache lines

When data is brought into cache, more than 1 value is fetched at a time.

A cache line typically holds 64 or 128 consecutive bytes (8 or 16 floats).

L1 Cache might hold 1000 cache lines.

Cache miss occurs if the the value you need next is not in cache.

Cache lines

When data is brought into cache, more than 1 value is fetched at a time.

A cache line typically holds 64 or 128 consecutive bytes (8 or 16 floats).

L1 Cache might hold 1000 cache lines.

Cache miss occurs if the the value you need next is not in cache.

Another cache line will be brought from higher up the hierachy, and may displace some variables in cache.

Those cache lines will first have to be written back to memory.

Bottom line: Good to do lots of work on each set of data while in cache, before it has to be written back.

Organize algorithm for Temporal locality.

Spatial locality

Also good to organize algorithm so data that is consecutive in memory is used together when possible.

If data you need is scattered through memory, many cache lines will be needed and will contain data you don't need.

This is called spatial locality.

Multi-dimensional array storage

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

```
Apy = reshape(array([10,20,30,40,50,60]), (2,3))
Afort = reshape((/10,20,30,40,50,60/), (/2,3/))
```

Suppose the array storage starts at memory location 3401.

In Python or Fortran, the elements will be stored in the order:

```
loc 3401    Apy[0,0] = 10       Afort(1,1) = 10
loc 3402    Apy[0,1] = 20       Afort(2,1) = 40
loc 3403    Apy[0,2] = 30       Afort(1,2) = 20
loc 3404    Apy[1,0] = 40       Afort(2,2) = 50
loc 3405    Apy[1,1] = 50       Afort(1,3) = 30
loc 3406    Apy[1,2] = 60       Afort(2,3) = 60
```

Memory layout of 1000×3 array in Fortran

Memory location or offset of each array element:

$$\begin{bmatrix} 1 & 1001 & 2001 \\ 2 & 1002 & 2002 \\ 3 & 1003 & 2003 \\ 4 & 1004 & 2004 \\ 5 & 1005 & 2005 \\ \vdots & \vdots & \vdots \\ 999 & 1999 & 2999 \\ 1000 & 2000 & 3000 \end{bmatrix}$$

Looping over elements by column steps through memory sequentially (stride = 1)

Looping over elements by row does not.
stride $= 1000$ in this case since we jump ahead 1000 locations in memory with each step.

Spatial locality

Suppose A is $n \times n$ matrix,
D is $n \times n$ diagonal matrix with diagonal elements d_i.

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Python?? Same number of flops!

```
for i in range(n):
    for j in range(n):
        b[i,j] = d[i] * a[i,j]
```

or

```
for j in range(n):
    for i in range(n):
        b[i,j] = d[i] * a[i,j]
```

Spatial locality

Suppose A is $n \times n$ matrix,
D is $n \times n$ diagonal matrix with diagonal elements d_i.

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Python?? Same number of flops!

```
for i in range(n):
    for j in range(n):
        b[i,j] = d[i] * a[i,j]
```

or

```
for j in range(n):
    for i in range(n):
        b[i,j] = d[i] * a[i,j]
```

Answer: First one faster in Python (but loops still slow!)

Spatial locality

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Fortran?? Same number of flops!

```
do i=1,n
    do j=1,n
        b(i,j) = d(i) * a(i,j)
    enddo; enddo
```

or

```
do j=1,n
    do i=1,n
        b(i,j) = d(i) * a(i,j)
    enddo; enddo
```

Spatial locality

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Fortran?? Same number of flops!

```
do i=1,n
    do j=1,n
        b(i,j) = d(i) * a(i,j)
    enddo; enddo
```

or

```
do j=1,n
    do i=1,n
        b(i,j) = d(i) * a(i,j)
    enddo; enddo
```

Answer: Second one faster in Fortran!

Array ordering — which loop is faster?

```
integer, parameter :: m = 4097, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
       a(i,j) = 0.d0
       enddo
    enddo

do j = 1,n
    do i=1,m
       a(i,j) = 0.d0
       enddo
    enddo
```

Array ordering — which loop is faster?

```
integer, parameter :: m = 4097, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
       a(i,j) = 0.d0
       enddo
    enddo

do j = 1,n
    do i=1,m
       a(i,j) = 0.d0
       enddo
    enddo
```

First: 0.72 seconds, Second: 0.19 seconds

Much worse if m is high power of 2

```
integer, parameter :: m = 4096, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
       a(i,j) = 0.d0
       enddo
    enddo

do j = 1,n
    do i=1,m
       a(i,j) = 0.d0
       enddo
    enddo
```

First: 2.4 seconds, Second: 0.19 seconds

More about cache

Simplied model of one level direct mapped cache.

32-bit memory address: 4.3×10^9 addresses

Suppose cache holds $512 = 2^9$ cache lines (9-bit address)

A given memory location cannot go anywhere in cache.
9 low order bits of memory address determine cache address.

More about cache

Simplied model of one level direct mapped cache.

32-bit memory address: 4.3×10^9 addresses

Suppose cache holds $512 = 2^9$ cache lines (9-bit address)

A given memory location cannot go anywhere in cache.
9 low order bits of memory address determine cache address.

For a memory fetch:

- Determine cache address, check if this holds desired words from memory.
- If so, use it.
- If not, check “dirty bit” to see if has been modified since load.
- If so, write to memory before loading new cache line.

Cache collisions

Return to example where matrix has $4096 = 2^{12}$ rows.

Cache line holds 64 bytes = 8 floats. $4096/8 = 512$ cache lines per column of matrix.

Loading one column of matrix will fill up cache lines $0, 1, 2, \ldots, 511$.

Second column will go back to cache line 0.
But all elements in cache have been used before this happens,
Prefetching can be done by optimizing compiler.

Cache collisions

Return to example where matrix has $4096 = 2^{12}$ rows.

Cache line holds 64 bytes = 8 floats. $4096/8 = 512$ cache lines per column of matrix.

Loading one column of matrix will fill up cache lines $0, 1, 2, \ldots, 511$.

Second column will go back to cache line 0.
But all elements in cache have been used before this happens,
    Prefetching can be done by optimizing compiler.

Worse — Going across the rows:

The first 8 elements of column 1 go to cache line 0.

The first 8 elements of column 2 also map to cache line 0.

Similarly for all columns. The rest of cache stays empty.

More about cache

If cache holds more lines:

1024 lines $\Longrightarrow$

first 8 bytes of column 1 go to cache line 0,
first 8 bytes of column 2 go to cache line 512,
first 8 bytes of column 3 go to cache line 0,
first 8 bytes of column 4 go to cache line 512.

Still only using 1/512 of cache.

More about cache

If cache holds more lines:

1024 lines $\Longrightarrow$

first 8 bytes of column 1 go to cache line 0,
first 8 bytes of column 2 go to cache line 512,
first 8 bytes of column 3 go to cache line 0,
first 8 bytes of column 4 go to cache line 512.

Still only using 1/512 of cache.

In practice cache is often set associative: small number of cache addresses for each memory address.

Padding

Matrix dimensions that are high powers of 2 should usually be avoided.

Even though natural for some algorithms such as FFTs

May be worth declaring larger arrays and only using part of it.

Code optimization

Basic considerations like memory layout should always be kept in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.

Code optimization

Basic considerations like memory layout should always be kept in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.
- Some optimizations not worth spending time on.
- Often best to first get code working properly and then determine whether optimization is necessary.
  "Premature optimization is the root of all evil" (Don Knuth)

Code optimization

Basic considerations like memory layout should always be kept in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.
- Some optimizations not worth spending time on.
- Often best to first get code working properly and then determine whether optimization is necessary.
  "Premature optimization is the root of all evil" (Don Knuth)
- If so, determine which parts of code need to be improved and spend effort on these sections. (Tools such as gprof)
- Use optimized software such as BLAS, LAPACK.