Lecture 15 – Performance Models and Tuning

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Performance Model Development--Dense Matrix-Vector Multiply
- Latency Based Execution Time Model (Important for Lab 08)
- Serial Architecture Performance Tuning
- (Time Permitting) Advanced Optimizations
    - What Optimized BLAS actually does

Learning Objectives: By the end of Today’s Lecture you should be able to

- (*Knowledge*) describe what algorithmic properties influence an algorithm’s performance
- (*Knowledge*) describe how hardware properties influence performance
- (*Skill*) develop and analyze a simple performance model to predict the performance of a low-level algorithm/computational kernel
- (*Knowledge*) be able to describe general techniques to “tune” the performance of code

Review of Lecture 14 Concepts

Motivation for Performance

- We can run more simulations in less time, and therefore get results quicker, learn quicker, publish quicker, graduate quicker, etc.
- While some common algorithms have been optimized, not every algorithm has, and perhaps for your problem/algorithm this is an area of research.
  - However, many of the same techniques can be employed for various algorithms.

Matrix Size	myDGEMM	OpenBLAS	MKL
100	0.001	0.064	0.002
500	0.068	0.069	0.007
1000	0.559	0.291	0.034
2000	6.160	0.749	0.231
4000	50.341	2.822	1.682

30x!

What is performance?

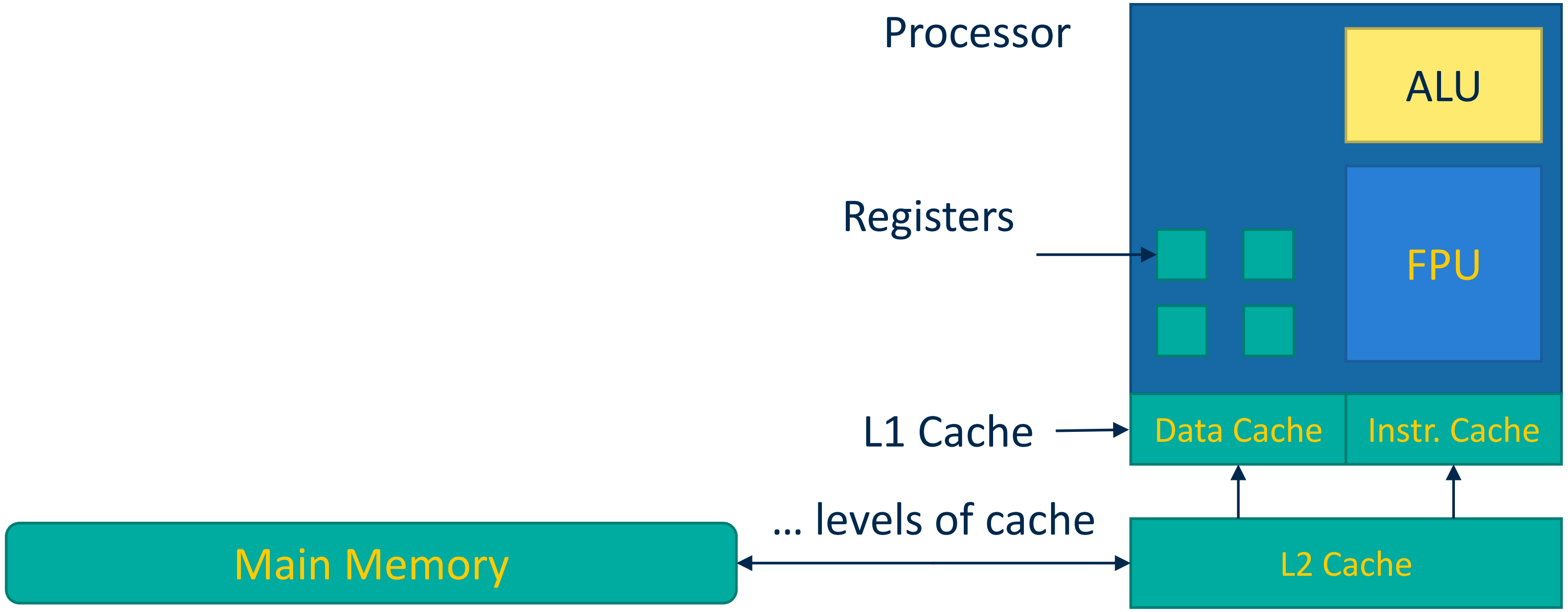
What does it mean for a code to be fast?

- The real metric: Time
- Derived metrics
  - *FLOPS* = FLoating Point OPerations per Second
  - *Bandwidth* = data per unit time (sort of like a flow rate)
  - *Latency* = Minimum time for data to travel from point A to point B
- Theoretical Peak Performance
  - *Very* difficult to achieve in practice
  - Can be computed from hardware specs
- Do things efficiently in time

How do you get fast code?

- First: Choose the right algorithm
- Second: Understand how to express that algorithm in the programming language
- Third: Understand how the source code will get mapped to the hardware
- Fourth: Tune the code to the hardware
- A lot of this can be done with pen and paper

Things to keep in mind about architecture



	Register	L1	L2	L3	DRAM	Disk	Tape
Size	< 1 KB	~1KB	1 MB	10’s MB	1-100’s GB	TB	PB
Speed	< 1ns	<1 ns	~1 ns	~1-10 ns	10-100 ns	10 ms	~10s

Fundamental Performance Model Concept

Execution time = time to perform arithmetic + *<u>time to move data</u>*

Understanding Performance

Peak Performance

- Example: Intel Haswell
  - What is the maximum FLOPs per cycle?
    - Need to look at SIMD information on processor
    - If we have AVX it supports a 256-bit vector so it can operate on 4 doubles
  - Does it support a fused multiply add (FMA instruction)?
    - Yes, so the chip can execute 4 FMA instructions (8 FLOPs) at once
  - How many vector units does it have?
    - Apparently it has 2 vector units… so now we’re at 16 FLOPs at once
  - How many cycles to execute an FMA instruction (which is two operations)?
    - Not always easy to find… common to assume 1 cycle.
      - However there may be other limiting factors such as latency (5 cycles in this case)
  - What is the clock speed (cycles per second)? 2.50 GHz
    - Well with AVX it appears to be 2.1 GHz
  - How many cores does it have? 12
- 16 FLOPs/cycle × 2.5e9 cycles/second × 12 cores = **480 GFLOPS**
  - 16 FLOPs ÷ 5 cycle × 2.1e9 cycles/second × 12 cores = **80 GFLOPS**
- Another derived metric is fraction of theoretical peak

Two Lessons from this:

1. This should be an easy calculation
2. Finding the right information can get quite complicated

3. Best to provide references e.g. document or presentation from the manufacturer

Algorithm Performance

- Not all algorithms are created equally
  - e.g. Big-O notation $O(n^2)$ vs $O(n \log n)$
- Not all implementations (algorithms really) are created equally
  - Can have "same" implementations with vastly different performance
- Very few algorithms allow you to achieve sustained performance at a significant fraction of the theoretical peak
- Let's go through an example

Simple Performance Model

- Assume just 2 levels of memory in hierarchy: fast memory and slow memory
- Consider a model to predict execution time for some set of operations

$$T = Ft_F + t_M L$$

F= # of FLOPs
L=# of loads
t_F = time for flop
t_M= time for memory load/store
T = execution time

- Minimum possible time is Ft_F when all data is in fast memory
- Can also rewrite as

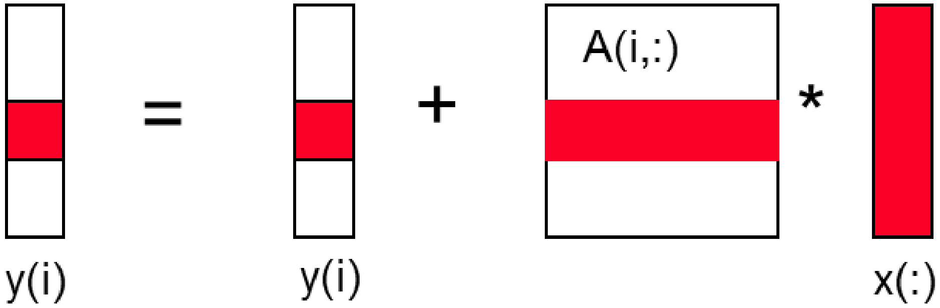
$$T = Ft_F\left(1+\frac{t_M}{t_F}\frac{L}{F}\right)$$

t_M/t_F is machine balance → key to machine efficiency
Generally $t_F \ll t_M$

$F/L=q$ is computational efficiency → key to algorithm efficiency
Larger q means time is closer to minimum.
$q \geq t_M / t_F$ to get at least *half* of the peak speed!

Matrix-Vector Multiply

```
{implements y = y + A*x}
for i = 1:n
          for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



Analysis of Matrix-Vector Multiply

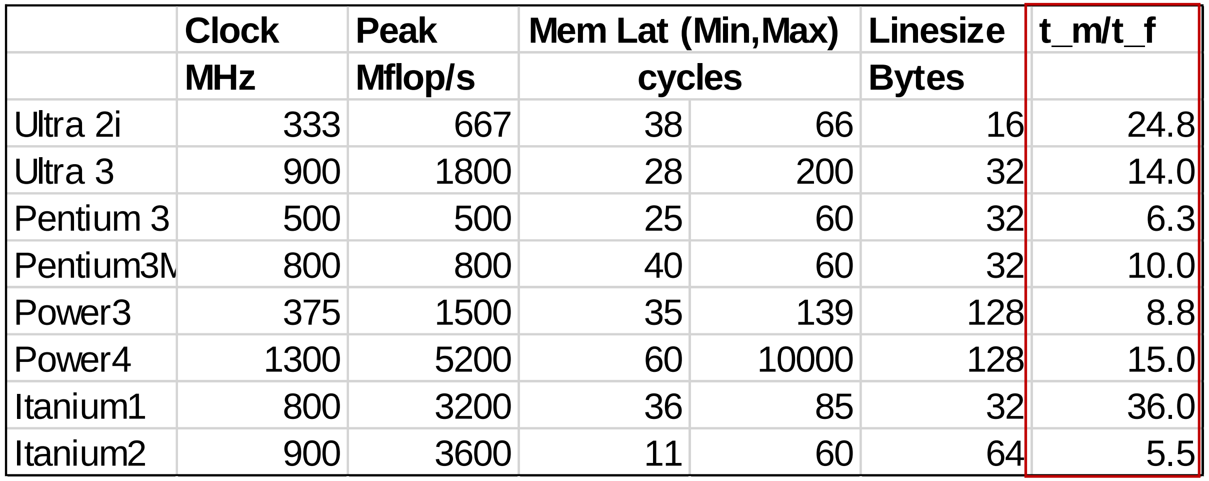
```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
        {read row i of A into fast memory}
        for j = 1:n
             y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- Assume all data for variables starts in slow memory
  - L = number of slow memory loads & stores
  - F = number of arithmetic operations
  - q =

Plug-n-Chug

- Some real data
  - From Flux t_M =8 ns (assuming L3 access time) and assume t_F = 0.3 ns (1 cycle)

$$T = 2n^2 t_F \left(1 + \frac{t_M}{t_F}\frac{1}{2}\right)$$



	Clock	Peak	Mem Lat (Min,Max)		Linesize	t_m/t_f
	MHz	Mflop/s	cycles		Bytes	
Ultra 2i	333	667	38	66	16	24.8
Ultra 3	900	1800	28	200	32	14.0
Pentium 3	500	500	25	60	32	6.3
Pentium3M	800	800	40	60	32	10.0
Power3	375	1500	35	139	128	8.8
Power4	1300	5200	60	10000	128	15.0
Itanium1	800	3200	36	85	32	36.0
Itanium2	900	3600	11	60	64	5.5

Table B.1 and B.2 from R. Vuduc Dissertation: http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

machine balance
(q must be at least this for ½ peak speed)

Generalization of Performance Models

- Latency based execution time model for “Single Processor”

$$T = Ft_F + \alpha_1 L + \sum_{j=1}^{\kappa-1}\left(\alpha_{j+1} - \alpha_j\right)M_j + \left(\alpha_{mem} - \alpha_\kappa\right)M_\kappa$$

F= # of FLOPs
L=# of loads
α = cache access latency
M = cache misses
T = execution time

	Register	L1	L2	L3	DRAM	Disk	Tape
Size	< 1 KB	~1KB	1 MB	10’s MB	1-100’s GB	TB	PB
Speed	< 1ns	<1 ns	~1 ns	~1-10 ns	10-100 ns	10 ms	~10s

- Most generally when dealing with complex kernels

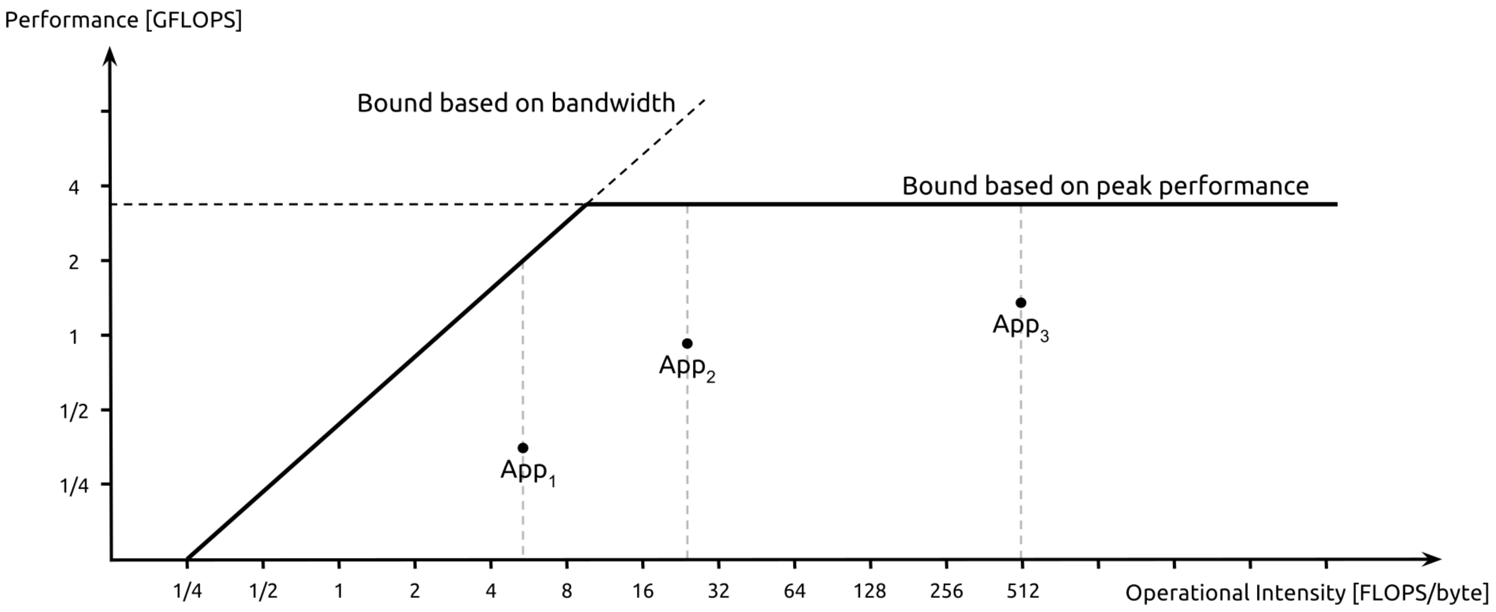
$$T = \sum_i N_i t_i$$

N_i = Number of operations of type i
t_i = time to execute operation of type i

Roofline Models

- Visual representation of performance relating arithmetic intensity (q) and hardware performance limits

$$q = \frac{F}{L}$$



Optimization & Tuning

Things people say about Optimization

- "We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil***. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will *be wise to look carefully at the critical code; but only after that code has been identified*"
    - Donald Knuth
- "You're bound to be unhappy if you optimize everything"
    - Donald Knuth
- "The best optimization you will ever have is to have your program go from not working to working"
    - paraphrasing

Common Optimization Techniques

- Before you program
  - Choose the best algorithm
    - e.g. choose known fastest converging algorithms or algorithms with asymptotically small operation counts
  - Choose the best way to express this algorithm in a programming language
    - Perform algebra to minimize operations or minimize memory traffic and communication
    - Design data structures around computational kernels & maximize cache locality
- As you are programming
  - compiler optimization flags
  - choose best operators (remove unnecessary FLOPs)
  - loop unrolling (pipelining & vectorization)
  - remove conditionals (lets compiler optimize loops better)
  - function tabulation (remove unnecessary FLOPs)
  - Mixed Precision

Compiler flags

GCC compiler option	Meaning
`-O0`	Reduce compilation time and make debugging produce the expected results. **This is the default**.
`-O1`	compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. (favors size of executable)
`-O2`	Performs nearly all supported optimizations that do not involve a space-speed tradeoff. Includes all `-O1` optimizations
`-O3`	Highest level of optimization. Includes all `-O2` optimizations
`-Ofast`	Disregard strict standards compliance. `-Ofast` enables all `-O3` optimizations. It also enables optimizations that are not valid for all standard-compliant programs.
`-Og`	Optimizations safe for debugging
`-fipa-pta`	Perform interprocedural pointer analysis and interprocedural modification and reference analysis. This option can cause excessive memory and compile-time usage on large compilation units. It is not enabled by default at any optimization level.
`-funsafe-math-optimizations`	Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards.

Choosing Data Structures and Loop Ordering

- Always make loops “stride-1” access to achieve best cache performance
  - Note this is not always possible for every algorithm.
- Looping structure should determine order of indexes in your variables.

Fortran loops should index “in-out”

```
DO k=1,n
  DO j=1,n
    DO i=1,n
      A(i,j,k)
     in ——→ out
    ENDDO
  ENDDO
ENDDO
```

C/C++ loops should index “out-in”

```
for( i=0; i<n; i++ ) {
  for( j=0; j<n; j++) {
    for( k=0; k<n; k++ ) {
      A[i][j][k]
      out ←—— in
    }
  }
}
```

Operator Choice

- Avoid exponentiation
  - Polynomial evaluation
    - e.g. Correlations for material properties, Semi-empirical models for coefficients

$$P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

$$P(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + a_4 x)))$$

```
t2=t**2.0  !slowest
t2=t**2    !slow
t2=t*t     !fastest
```

When its unavoidable

```
c=a**b
c=EXP(b*LOG(x))  !may be faster
```

- Division is allegedly more expensive than multiplication

```
DO i=1,n
  a(i)=b(i)/c
ENDDO
```

```
rc=1.0d0/c
DO i=1,n
  a(i)=b(i)*c
ENDDO
```

Expose Independent Operations

- Hide instruction latency
  - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  - Balance the instruction mix (e.g. what functional units are available?)

```
f1 = f5 * f9;
f2 = f6 + f10;
f3 = f7 * f11;
f4 = f8 + f12;
```

Exploit Multiple Registers

- Reduce demands on memory bandwidth by pre-loading into local variables

also: `register float f0 = …;`

```
while( … ) {
   *res++ = filter[0]*signal[0]
          + filter[1]*signal[1]
          + filter[2]*signal[2];
   signal++;
}
```

```
float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while( … ) {
   *res++ = f0*signal[0]
          + f1*signal[1]
          + f2*signal[2];
   signal++;
}
```

Loop Unrolling

- Compilers are not necessarily very good (or not as good as we’d like sometimes) at interpreting how to optimize loops
  - So compilers will do this if right flags are provided and loops are “clear” enough to compiler

```
subroutine lngth1(n,a,s)
  integer :: n
  real(8) :: s,a(n)
  integer :: i
  s=0.d0
  do i=1,n
    s=s+a(i)*a(i)
  enddo
endsubroutine
```

```
!works correctly only if the array size is a multiple of 4
subroutine lngth4(n,a,s)
  integer :: n
  real(8) :: s,a(n)
  integer :: i
  real(8) :: t1,t2,t3,t4

  t1=0.d0; t2=0.d0; t3=0.d0; t4=0.d0
  do i=1,n-3,4
    t1=t1+a(i)*a(i)
    t2=t2+a(i+1)*a(i+1)
    t3=t3+a(i+2)*a(i+2)
    t4=t4+a(i+3)*a(i+3)
  enddo
  s=t1+t2+t3+t4
endsubroutine
```

Unrolled to a depth of 4

Loop Unrolling (2)

- Exploits vector instructions and pipelining
- Cannot be done to arbitrary size
  - Registers will get overloaded
  - Size should be “register-blocked” or “cache-blocked”.
- Can use *padding* to avoid remainder loops.

```
subroutine lngth4(n,a,s)
  integer :: n
  real(8) :: s,a(n)
  integer :: i
  real(8) :: t1,t2,t3,t4,tr

  t1=0.d0; t2=0.d0; t3=0.d0; t4=0.d0; tr=0.d0
  do i=1,n-3,4
    t1=t1+a(i)*a(i)
    t2=t2+a(i+1)*a(i+1)
    t3=t3+a(i+2)*a(i+2)
    t4=t4+a(i+3)*a(i+3)
  enddo
  do j=n-MOD(n,4)+1,n
    tr=tr+a(j)*a(j) !in practice need “remainder”
  enddo
  s=t1+t2+t3+t4+tr
endsubroutine
```

Remove conditionals from loops

- Loops that have branching constructs are usually not optimized by compiler
  - Set all even indices to 0 and all odd indices to 1

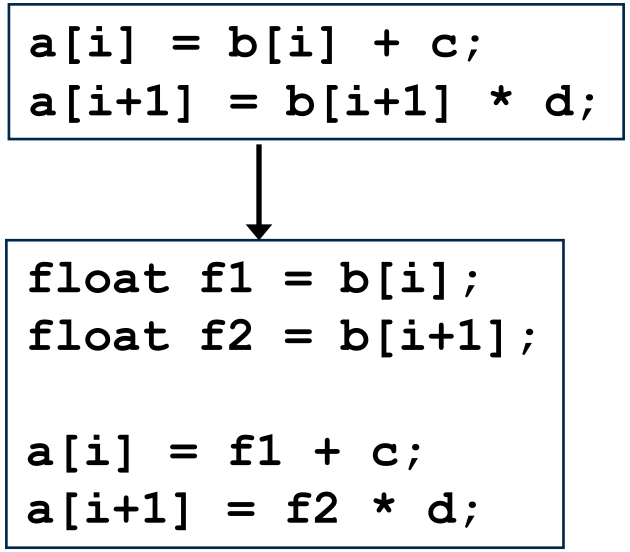
```
DO i=1,SIZE(a)
  IF(MOD(i,2 == 1)) THEN
    a(i)=1
  ELSE
    a(i)=0
  ENDIF
ENDDO
```

```
DO i=1,SIZE(a),2
  a(i)=1
ENDDO

DO i=2,SIZE(a),2
  a(i)=0
ENDDO
```

```
DO i=1,SIZE(a)-1,2
  a(i)=1
  a(i+1)=0
ENDDO
IF(MOD(SIZE(a),2) == 1) &
  a(SIZE(a))=1
```

Removing False Dependencies

- Using local variables, reorder operations to remove false dependencies



```
a[i] = b[i] + c;
a[i+1] = b[i+1] * d;
```

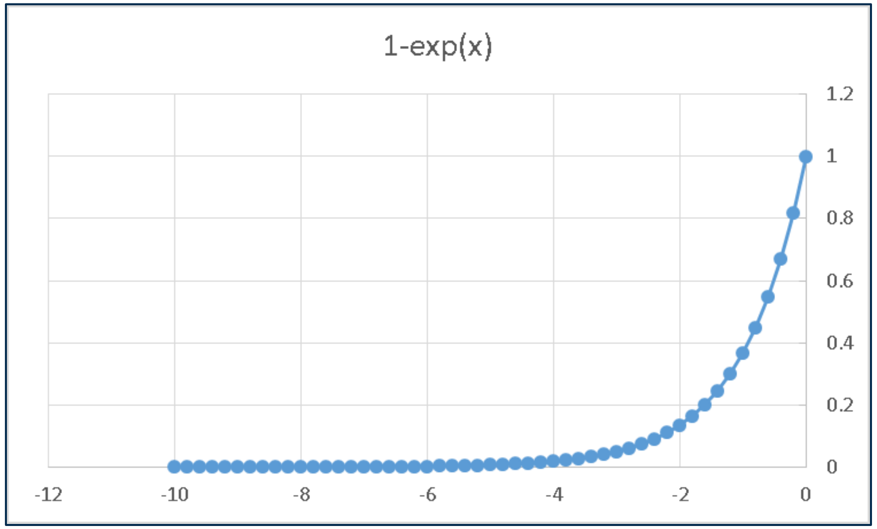
```
float f1 = b[i];
float f2 = b[i+1];

a[i] = f1 + c;
a[i+1] = f2 * d;
```

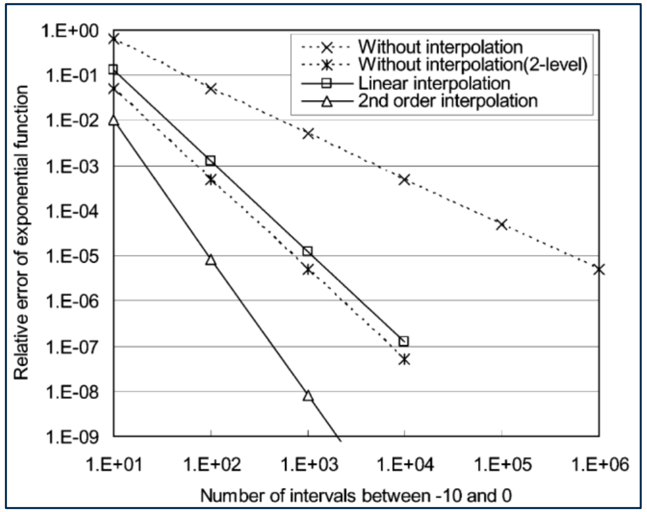
false read-after-write hazard
between a[i] and b[i+1]

- With some compilers, you can declare a and b unaliased.
  - Done via “restrict pointers”, compiler flag, or pragma.

Function tabulation

- Some special functions (exponential, logarithm, gamma function, error function, etc.)
  - Require many FLOPs to evaluate to double precision.
- Tabulate function and linearly interpolate result
  - Introduces interpolation error.
  - Error is generally proportional to of table size
  - If evaluating the table A LOT, want table to be small enough to fit into cache
- In some cases, we can accept interpolation error because we do not know physical value to double precision (e.g. 15 digits) accuracy.

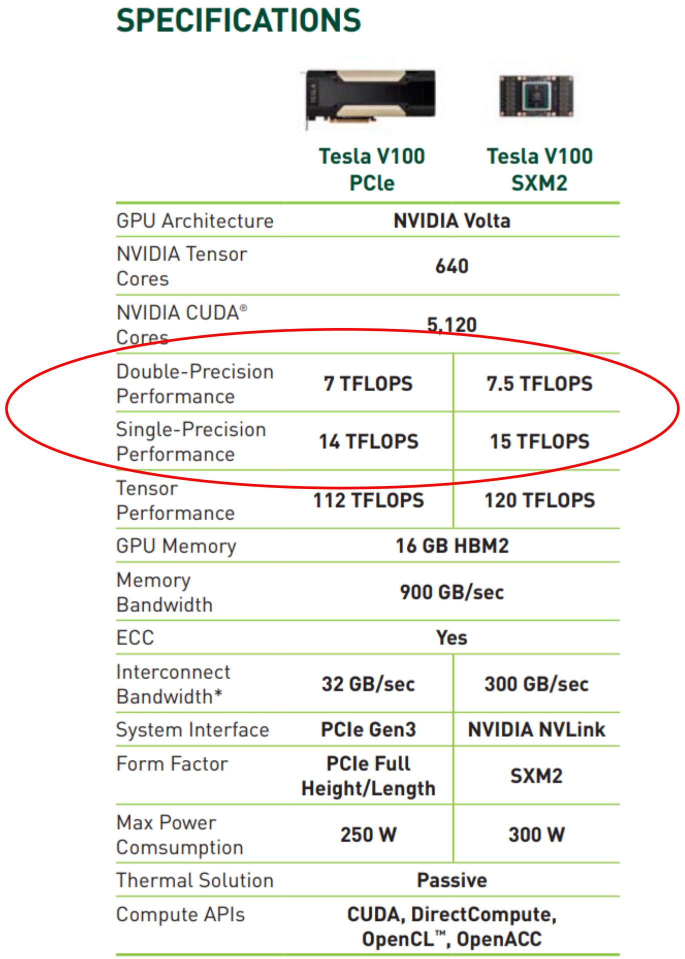




Mixed Precision

- Not all data should be expressed as double precision
  - Using single precision data means you are “moving” less data.
  - Can reduce storage required coefficients (if you have a lot of data here)
- Target opportunities for introducing single precision:
  - Preconditioners in Krylov methods
  - Coefficients (when based on experimental measurement) are not necessarily known to double precision
    - Geometry data
    - Material compositions
- Avoid the pitfall of losing robustness in iterative methods by making iterates single precision. Iterates and coefficient matrices should always be double precision.

GPU Specs



SPECIFICATIONS

	Tesla V100 PCIe	Tesla V100 SXM2
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.5 TFLOPS
Single-Precision Performance	14 TFLOPS	15 TFLOPS
Tensor Performance	112 TFLOPS	120 TFLOPS
GPU Memory	16 GB HBM2	
Memory Bandwidth	900 GB/sec	
ECC	Yes	
Interconnect Bandwidth*	32 GB/sec	300 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink
Form Factor	PCIe Full Height/Length	SXM2
Max Power Comsumption	250 W	300 W
Thermal Solution	Passive	
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC	

More advanced techniques

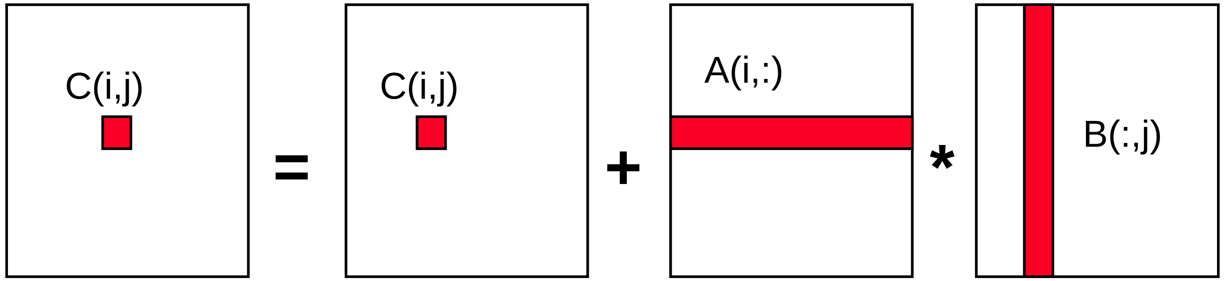
- Memory blocking
- Cache oblivious ordering
- Communication Avoiding Algorithms (State of the Art)

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

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

Naïve 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}
```

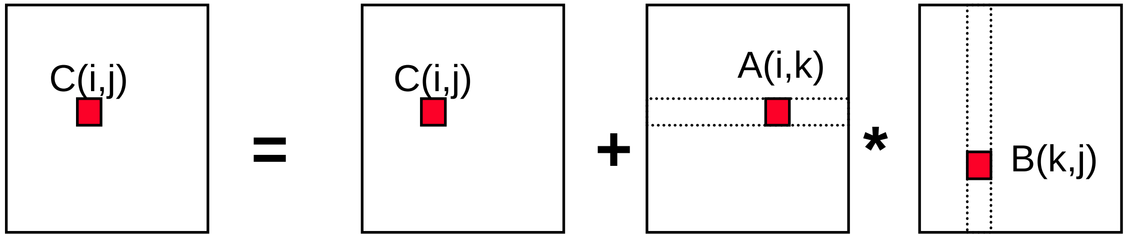


Memory Blocking (Tiling)

- Idea: improve computational intensity and temporal locality of data.

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}
```



Analysis of Blocked 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 $q = f / m = 2n^3 / ((2N + 2) * n^2)$

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

So we can improve performance by increasing the blocksize b

Can be much faster than matrix-vector multiply (q=2)

Larger block size = more efficient

Limit: All three blocks from A,B,C must fit into fast memory

Assume fast memory size M_{fast}

$3b^2 \leq M_{fast}$, so $q \approx b \leq (M_{fast}/3)^{1/2}$

Cache Oblivious Algorithms

- Typically implemented as recursive algorithms and recursive data structures
- Tiled algorithm requires finding good block size (will depend on hardware)
- Cache Oblivious Algorithms offer an alternative
  - Idea is to order things in memory to minimize latency with multiple levels of memory hierarchy.
  - Make use of Space Filling Curves

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}$$

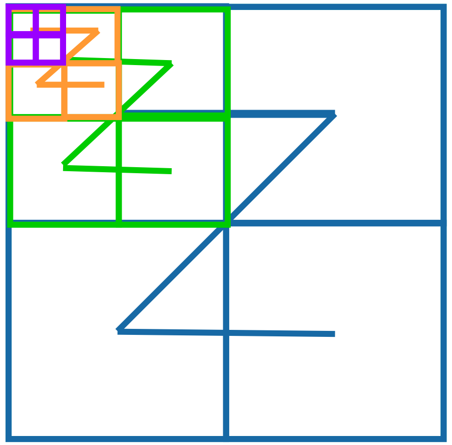
$$= \begin{pmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{pmatrix}$$

Advantages:

- the recursive layout works well for any cache size

Disadvantages:

- The index calculations to find A[i,j] are expensive
- Implementations switch to column-major for small sizes



Z-Order Space Filling Curve

Summary of Serial Optimizations

- Details of machine are important for performance
  - Processor and memory system (not just parallelism)
  - What to expect? Use understanding of hardware limits
- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case
- There is parallelism hidden within processors
  - Pipelining, SIMD, etc

- Data locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby that recently used
- Can rearrange code/data to improve locality
  - Goal: minimize communication = data movement
- Performance intensive code should be written clearly for compiler (not for humans)