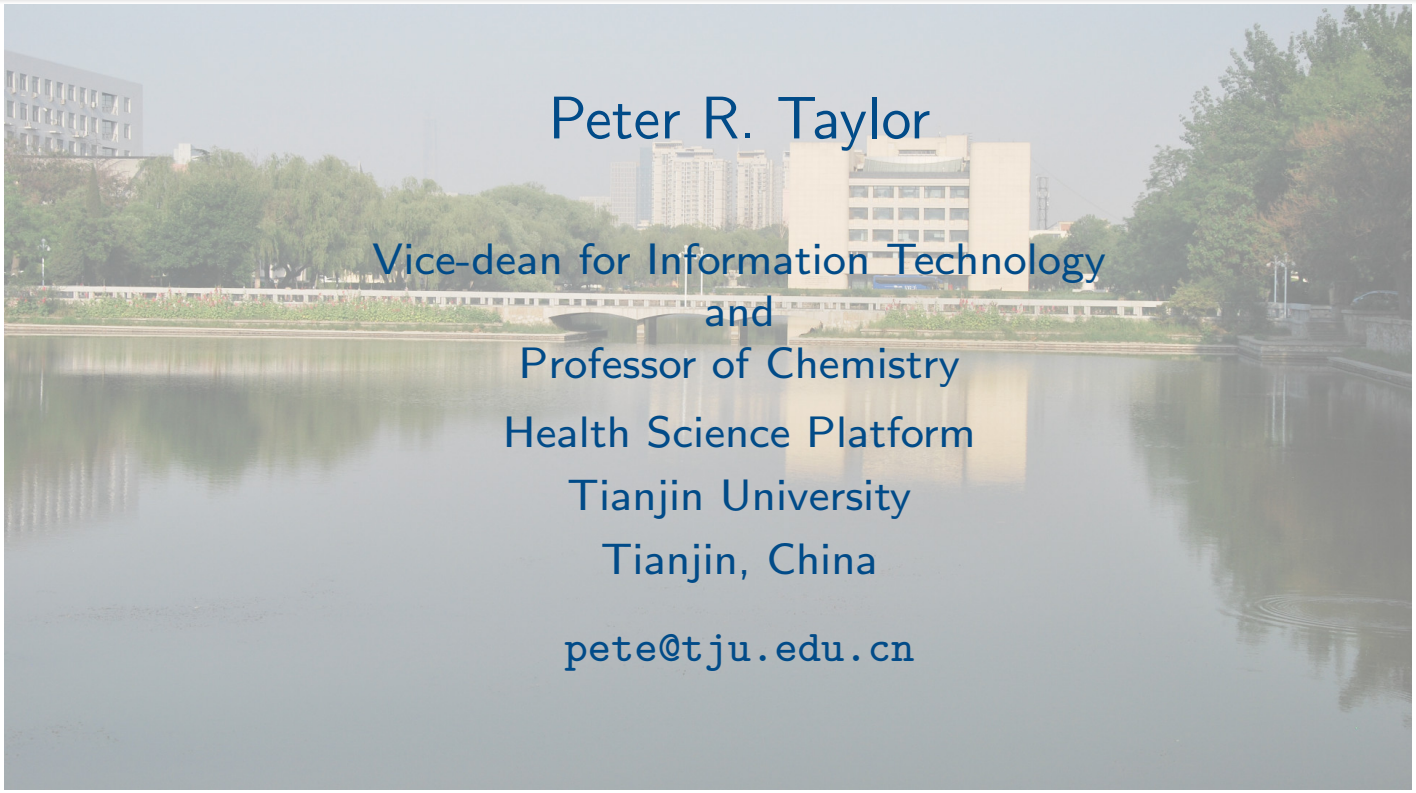
Computational Linear Algebra



Peter R. Taylor

Vice-dean for Information Technology
and
Professor of Chemistry

Health Science Platform
Tianjin University
Tianjin, China

pete@tju.edu.cn

November 18–22, 2019


Molecules
Medicine
Biomedical Engineering
Life Science
Public Health Policy
Informatics


TIANJIN UNIVERSITY (PEIYANG UNIVERSITY)
1895
天津大学


Molecules
Medicine
Biomedical Engineering
Life Science
Public Health Policy
Informatics


TIANJIN UNIVERSITY (PEIYANG UNIVERSITY)
1895
天津大学


Molecules
Medicine
Biomedical Engineering
Life Science
Public Health Policy
Informatics


TIANJIN UNIVERSITY (PEIYANG UNIVERSITY)
1895
天津大学

Computational linear algebra

- Some history; processors and vendor roadmaps: current designs, multicore issues, GPUs, TPUs, FPGAs. Caching and virtual memory
- Review of (mostly real) linear algebra: vectors and matrices, determinants, matrix/vector operations (BLAS), block matrix operations.
- Programming: languages, libraries (BLAS, LAPACK, etc.), OpenMP/OpenACC, MPI, CUDA, Scalapack/BLACS.
- Standard methods: matrix multiplication, matrix transformations, Gaussian elimination, matrix diagonalization, singular value decomposition, Cholesky factorization.

Computational linear algebra

- Performance measurement: strategies, code optimization and debugging packages.
- Optimization of functions: first- and second-order methods, constraints, trust-region methods.
- Large systems: linear and nonlinear equations, diagonalization
- Factorization methods (Cholesky, resolution of the identity) in computational chemistry.

Recommended reading

- Golub and van Loan "Matrix Computations" (Johns Hopkins, 4th ed).
- Strang "Linear Algebra and its Applications" (Brooks Cole, 4th ed [or higher now?]).
- Fletcher "Practical Methods of Optimization" (Wiley, 2nd ed).

Ancient history

- 1940s/1950s technology: vacuum tubes, mercury delay lines...
- 1960s: semiconductors — ferrite core memory, tapes (sequential storage).
- 1970s: integrated circuits ("chips"), semiconductor memory, disk/drum storage (random access storage).

Ancient history

- Small memories, slow CPUs.

Ancient history

- Small memories, slow CPUs. *Really small memories*, like 256KB.
- Programming priority 1: use as little memory as possible.
- Programming priority 2: do as few operations as possible.

Ancient history

- Small memories, slow CPUs. *Really small memories*, like 256KB.
- Programming priority 1: use as little memory as possible.
- Programming priority 2: do as few operations as possible. Especially arithmetic operations.

Ancient history

- Small memories, slow CPUs. *Really small memories*, like 256KB.
- Programming priority 1: use as little memory as possible.
- Programming priority 2: do as few operations as possible. Especially arithmetic operations. *Especially* floating-point (FP) arithmetic operations...

Ancient history

- Small memories, slow CPUs. *Really small memories*, like 256KB.
- Programming priority 1: use as little memory as possible.
- Programming priority 2: do as few operations as possible. Especially arithmetic operations. *Especially* floating-point (FP) arithmetic operations...
- Seymour Cray and the revolution...

Seymour Cray

- UNIVAC to Control Data: sole interest was *creating the fastest machine around*.
- CDC6600: pathway to CDC7600. Pipelining
- E.g., FP add
  - Load operands into arithmetic unit
  - Unpack reals and shift to match exponents
  - Add
  - Pack to standard FP form
  - Store result
- Old-skool: complete one add, then start next.
- Cray: multi-stage CPU pipeline, *start next operation every clock tick.*

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).
- In practice, 7 MFLOPS excellent, 10 MFLOPS amazing...

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).
- In practice, 7 MFLOPS excellent, 10 MFLOPS amazing...
- Contention for memory, inability to connect adds and multiplies, small instruction stack.

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).
- In practice, 7 MFLOPS excellent, 10 MFLOPS amazing...
- Contention for memory, inability to connect adds and multiplies, small instruction stack.
- Cray's brainwave: the CDC 8600!

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).
- In practice, 7 MFLOPS excellent, 10 MFLOPS amazing...

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).
- In practice, 7 MFLOPS excellent, 10 MFLOPS amazing...
- Contention for memory, inability to connect adds and multiplies, small instruction stack.

CDC7600

- Design processing units to produce one result every clock tick.
- Good idea, but not achieved. Potentially (27.5 nS clock) 36 MFLOPS (million floating point operations per second).
- In practice, 7 MFLOPS excellent, 10 MFLOPS amazing...
- Contention for memory, inability to connect adds and multiplies, small instruction stack.
- Cray's brainwave: the CDC 8600!

The "CDC8600"

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.

The "CDC8600"

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.

The "CDC8600"

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.
- Clock 12.5 ns: 80 MFLOPS add or multiply, 160 MFLOPS *both*! (Division more complicated. . .)

The "CDC8600"

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.
- Clock 12.5 ns: 80 MFLOPS add or multiply, 160 MFLOPS *both*! (Division more complicated...)
- Control Data couldn't/wouldn't believe it...

The "CDC8600"

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.
- Clock 12.5 ns: 80 MFLOPS add or multiply, 160 MFLOPS *both*! (Division more complicated...)
- Control Data couldn't/wouldn't believe it...
- Cray left (on good terms) to use his design for his own company.

The CRAY-1

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.
- Clock 12.5 ns: 80 MFLOPS add or multiply, 160 MFLOPS *both*! (Division more complicated. . .)

The CRAY-1

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.
- Clock 12.5 ns: 80 MFLOPS add or multiply, 160 MFLOPS *both*! (Division more complicated. . .)
- Specified in the instruction set, and implemented in libraries provided to the user.

The CRAY-1

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.
- Clock 12.5 ns: 80 MFLOPS add or multiply, 160 MFLOPS *both*! (Division more complicated. . .)
- Specified in the instruction set, and implemented in libraries provided to the user.
- First machine to achieve (more or less. . .) design/market specifications.

The CRAY-1

- Design processing units to produce one result every clock tick and ensure they feed one another so that is sustainably achieved.
- Every clock an add/subtract result, *and* and a multiply.
- Add unit and multiply unit *chained* — one caould be fed into the other: *still* one result per clock period.
- Clock 12.5 ns: 80 MFLOPS add or multiply, 160 MFLOPS *both*! (Division more complicated. . .)
- Specified in the instruction set, and implemented in libraries provided to the user.
- First machine to achieve (more or less. . .) design/market specifications. New world for computational science!

The CRAY-1

- So we get 160 MFLOPS, instead of 7 to 10 (CDC 7600): factor of 10+!

The CRAY-1

- So we get 160 MFLOPS, instead of 7 to 10 (CDC 7600): factor of 10+! YES!!!

The CRAY-1

- So we get 160 MFLOPS, instead of 7 to 10 (CDC 7600): factor of 10+! YES!!!
- Well, no...
- Some codes factor of 4 or 5. Many factor of 2.

The CRAY-1

- So we get 160 MFLOPS, instead of 7 to 10 (CDC 7600): factor of 10+! YES!!!
- Well, no...
- Some codes factor of 4 or 5. Many factor of 2. They were written wrong!

The CRAY-1

- So we get 160 MFLOPS, instead of 7 to 10 (CDC 7600): factor of 10+! YES!!!
- Well, no...
- Some codes factor of 4 or 5. Many factor of 2. They were written wrong!
- Had to *abandon* old ideas about minimizing operation count.

The CRAY-1

- So we get 160 MFLOPS, instead of 7 to 10 (CDC 7600): factor of 10+! YES!!!
- Well, no...
- Some codes factor of 4 or 5. Many factor of 2. They were written wrong!
- Had to *abandon* old ideas about minimizing operation count.
- Had to *abandon* old ideas about memory use!

The CRAY-1

- So we get 160 MFLOPS, instead of 7 to 10 (CDC 7600): factor of 10+! YES!!!
- Well, no...
- Some codes factor of 4 or 5. Many factor of 2. They were written wrong!
- Had to *abandon* old ideas about minimizing operation count.
- Had to *abandon* old ideas about memory use!
- Had to ensure that all compute-intensive work was implemented as simple vector loops or as matrix operations.

Vector computing

- Over a period of several years, most codes were rewritten as "vector codes". At least, those that could be.

Vector computing

- Over a period of several years, most codes were rewritten as "vector codes". At least, those that could be.
- Not only got top performance on the Cray, but usually got significant performance improvement on the older computers!

Vector computing

- Over a period of several years, most codes were rewritten as "vector codes". At least, those that could be.
- Not only got top performance on the Cray, but usually got significant performance improvement on the older computers!
- Rewriting led to cleaner, simpler code that compilers could more easily optimize, and which usually took more advantage of optimized vendor libraries.

Vector computing

- Over a period of several years, most codes were rewritten as "vector codes". At least, those that could be.
- Not only got top performance on the Cray, but usually got significant performance improvement on the older computers!
- Rewriting led to cleaner, simpler code that compilers could more easily optimize, and which usually took more advantage of optimized vendor libraries.
- Many vendors followed: IBM 3090VF, CDC Cyber205, and Cray went on to produce the multiprocessor X–MP, the "huge" memory (2GB in 1985) Cray 2, and follow-ons to both.

Nonspecialized vector computing

Nonspecialized vector computing

- Designing ever-faster specialized super-expensive hardware was never sustainable.

Nonspecialized vector computing

- Designing ever-faster specialized super-expensive hardware was never sustainable.
- Much of the functionality was incorporated into mainstream processors (e.g., Intel's AVX) and supported by their libraries.

Nonspecialized vector computing

- Designing ever-faster specialized super-expensive hardware was never sustainable.
- Much of the functionality was incorporated into mainstream processors (e.g., Intel's AVX) and supported by their libraries.
- Physics is against faster and faster processors: power draw goes as (frequency)^3.

Nonspecialized vector computing

- Designing ever-faster specialized super-expensive hardware was never sustainable.
- Much of the functionality was incorporated into mainstream processors (e.g., Intel's AVX) and supported by their libraries.
- Physics is against faster and faster processors: power draw goes as $(\text{frequency})^3$.
- Inevitably, then, processor speed plateaus and the only way to increase performance is to have *and use* more processors: parallel computing...

Parallel computing

Parallel computing

- Seemed we'd barely mastered vector computing. . .

Parallel computing

- Seemed we'd barely mastered vector computing. . . .
- Major complication. Vector computing was largely cleaning up and reimplementing existing algorithms. Parallel computing meant new algorithms needed.

Parallel computing

- Seemed we'd barely mastered vector computing. . . .
- Major complication. Vector computing was largely cleaning up and reimplementing existing algorithms. Parallel computing meant new algorithms needed.
- Some people gave up.

Parallel computing

- Seemed we'd barely mastered vector computing. . . .
- Major complication. Vector computing was largely cleaning up and reimplementing existing algorithms. Parallel computing meant new algorithms needed.
- Some people gave up.
- Some were incredibly inventive, especially coping with "non-uniform memory access": local vs remote memory.

Parallel computing

- Seemed we'd barely mastered vector computing. . .
- Major complication. Vector computing was largely cleaning up and reimplementing existing algorithms. Parallel computing meant new algorithms needed.
- Some people gave up.
- Some were incredibly inventive, especially coping with "non-uniform memory access": local vs remote memory.
- Many toolkits like Global Arrays. We will look at some later.

Parallel computing

- Seemed we'd barely mastered vector computing. . . .
- Major complication. Vector computing was largely cleaning up and reimplementing existing algorithms. Parallel computing meant new algorithms needed.
- Some people gave up.
- Some were incredibly inventive, especially coping with "non-uniform memory access": local vs remote memory.
- Many toolkits like Global Arrays. We will look at some later.
- Our task is *multilevel* parallelism: multicore processors and multiple processors. *Fine-* and *coarse-grained* parallelism.

Today and tomorrow...

Today and tomorrow...

- Multicore "standard" CPUs (x86_64), more and more cores.

Today and tomorrow...

- Multicore "standard" CPUs (x86_64), more and more cores.
- Low-power "standard" CPUs (ARM — buy a Raspberry Pi to get experience...).

Today and tomorrow...

- Multicore "standard" CPUs (x86_64), more and more cores.
- Low-power "standard" CPUs (ARM — buy a Raspberry Pi to get experience...).
- GPUs! nVIDIA killing both AMD and particularly Intel (Larrabee, Xeon Phi...).

Today and tomorrow...

- Multicore "standard" CPUs (x86_64), more and more cores.
- Low-power "standard" CPUs (ARM — buy a Raspberry Pi to get experience...).
- GPUs! nVIDIA killing both AMD and particularly Intel (Larrabee, Xeon Phi...).
- TPUs: are they of use to us?

Today and tomorrow...

- Multicore "standard" CPUs (x86_64), more and more cores.
- Low-power "standard" CPUs (ARM — buy a Raspberry Pi to get experience...).
- GPUs! nVIDIA killing both AMD and particularly Intel (Larrabee, Xeon Phi...).
- TPUs: are they of use to us?
- FPGAs: the way of the future.

Today and tomorrow...

- Multicore "standard" CPUs (x86_64), more and more cores.
- Low-power "standard" CPUs (ARM — buy a Raspberry Pi to get experience...).
- GPUs! nVIDIA killing both AMD and particularly Intel (Larrabee, Xeon Phi...).
- TPUs: are they of use to us?
- FPGAs: the way of the future.Always have been, always will be...

The day after tomorrow...

The day after tomorrow...

- Quantum computing?

The day after tomorrow...

- Quantum computing?
- Always just around the corner.

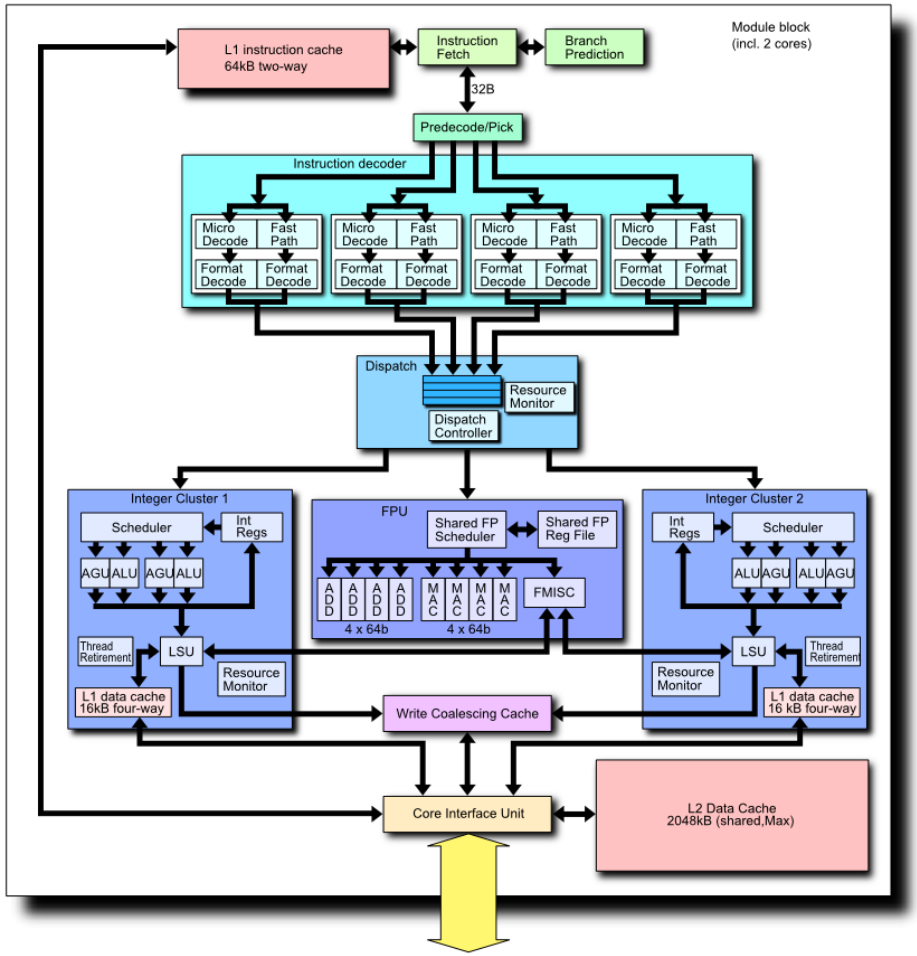
The day after tomorrow...

- Quantum computing?
- Always just around the corner.
- *Could* be a "game-changer".

The day after tomorrow...

- Quantum computing?
- Always just around the corner.
- *Could* be a "game-changer".
- We have to see...

Typical x86


Module block
(incl. 2 cores)
L1 instruction cache
64kB two-way
Instruction
Fetch
Branch
Prediction
32B
Predecode/Pick
Instruction decoder
Micro
Decode
Fast
Path
Format
Decode
Dispatch
Resource
Monitor
Dispatch
Controller
Integer Cluster 1
Scheduler
Int
Regs
AGU
ALU
Thread
Retirement
LSU
Resource
Monitor
L1 data cache
16kB four-way
FPU
Shared FP
Scheduler
Shared FP
Reg File
ADD
MAC
FMISC
4 x 64b
Integer Cluster 2
Write Coalescing Cache
Core Interface Unit
L2 Data Cache
2048kB (shared,Max)
L1 data cache
16 kB four-way

Points to note

Points to note

- Managing memory becomes very tricky: multiple cache levels, RAM, virtual memory; access from multiple cores.

Points to note

- Managing memory becomes very tricky: multiple cache levels, RAM, virtual memory; access from multiple cores.
- Cannot leave any of this to users: need appropriate firmware/hardware.

Points to note

- Managing memory becomes very tricky: multiple cache levels, RAM, virtual memory; access from multiple cores.
- Cannot leave any of this to users: need appropriate firmware/hardware.
- Need lightweight "threads" of execution as tasks: quite different from traditional UNIX/Linux `fork/exec`.

Points to note

- Managing memory becomes very tricky: multiple cache levels, RAM, virtual memory; access from multiple cores.
- Cannot leave any of this to users: need appropriate firmware/hardware.
- Need lightweight "threads" of execution as tasks: quite different from traditional UNIX/Linux `fork/exec`.
- "Hyperthreading" to maximize use of cores: workload-dependent.

Vectors

- Vector space $\{\mathbf{x}^k\}$

$$\alpha(\mathbf{x}^i + \mathbf{x}^j) = \alpha\mathbf{x}^i + \alpha\mathbf{x}^j.$$

  k-dimensional vector space.
- k need not be finite.
- Linear independence:

$$\sum_i \alpha_i \mathbf{x}^i = 0 \Rightarrow \alpha_i = 0 \,\forall\, i.$$

- Otherwise the set is *linearly dependent*.

Vectors

- Write as columns

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix}$$

  for a k-dimensional vector space.

- *Transpose* is a row vector

$$\mathbf{x}^T = (x_1\ x_2\ \cdots\ x_k)$$

Scalar product

- Two vectors $\mathbf{x}$ and $\mathbf{y}$:

$$\mathbf{x} \cdot \mathbf{y} \equiv \langle \mathbf{x} | \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \sum_p x_p y_p.$$

- $\mathbf{x} \cdot \mathbf{x} \geq 0$.
- *Orthonormal* vector space:

$$\mathbf{x}^i \cdot \mathbf{x}^j = \delta_{ij}.$$

Matrices

- Array of elements

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix}.$$

  m rows, n columns. $m \times n$ matrix.
- "Square matrix" if $m = n$.
- *Symmetric* if square and all $A_{pq} = A_{qp}$.

Matrix properties

- Consider as a set of n m-dimensional column vectors, or m n-dimensional row vectors.
- *Span*, *rank*, *kernel*.
- Matrix multiplication **AB**:

$$C_{pq} = \sum_r A_{pr} B_{rq},$$

  matrices must be *compliant*: **A** $m \times k$, **B** $k \times n$, so **C** $m \times n$.
- Holds also for vectors: scalar and *matrix* product.

Norms

- *Vector p-norms*

$$\|x\|_p = \left(\sum_k x_k^p \right)^{\frac{1}{p}}.$$

- $\|x\|_2 = \mathbf{x}^T \mathbf{x}$.
- $\|x\|_\infty = \max |x_i|$.
- *Cauchy-Schwarz* inequality

$$|\mathbf{x}^T \mathbf{y}| \leq \|x\|_2 \, \|y\|_2.$$

Special case of Hölder's inequality.

Norms

- *Matrix p-norms* can be defined analogous to vector p-norms, e.g.,

$$\|\mathbf{A}\|_\infty = \max |A_{ij}|.$$

- *Frobenius norm*

$$\|\mathbf{A}\|_F = \left(\sum_i \sum_j |A_{ij}|^2 \right)^{\frac{1}{2}}$$

Determinants

- Many definitions, e.g.,

$$\det(\mathbf{A}) \equiv |\mathbf{A}| = \sum_{j=1}^{n} (-1)^(j+1) A_{1j} \det(\mathbf{A}_{1j}),$$

  recursively for an $n \times n$ matrix $\mathbf{A}$.
- This is "expanding along the top row".
- $\det(\mathbf{A}_{1j})$ is a *minor* of $\mathbf{A}$, and $(-1)^(j+1) \det(\mathbf{A}_{1j})$ is *cofactor*, or signed minor.
- Key property: if $\det(\mathbf{A}) = 0$, the matrix $\mathbf{A}$ is *singular*: it has no inverse.

Determinants

- "Proper formula", attributed to Leibniz

$$\det(\mathbf{A}) = \sum_P \sigma_P \prod_i A_{iP(i)} \equiv \sum_P \sigma_P \prod_i A_{P(i)i},$$

where P runs over all permutations of the integers $1 \ldots n$, and σ_P is the *sign*, or *parity* of the permutation, according to whether it comprises an odd or even number of transpositions.

Determinants

- "Proper formula", attributed to Leibniz

$$\det(\mathbf{A}) = \sum_P \sigma_P \prod_i A_{iP(i)} \equiv \sum_P \sigma_P \prod_i A_{P(i)i},$$

  where P runs over all permutations of the integers $1 \ldots n$, and σ_P is the *sign*, or *parity* of the permutation, according to whether it comprises an odd or even number of transpositions.
- For anyone with a background in group theory, this is a projection operator for the *alternating*, or *antisymmetric* irreducible representation.
- Leads to other related quantities — consider costs for evaluation.

Block operations

- Recursive generalization of matrix operations. E.g., matrices **A** and **B** which have a block structure

$$\mathbf{A} = \left(\begin{array}{ccc|ccc} A_{11} & \cdots & A_{1l} & A_{1(l+1)} & \cdots & A_{1n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{k1} & \cdots & A_{kl} & A_{k(l+1)} & \cdots & A_{kn} \\ \hline A_{(k+1)1} & \cdots & A_{(k+1)l} & A_{(k+1)(l+1)} & \cdots & A_{(k+1)n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{m1} & \cdots & A_{ml} & A_{m(l+1)} & \cdots & A_{mn} \end{array}\right),$$

where for same-sized square blocks (not necessary) $2k = m$ and $2l = n$.

Block operations

- Write this as

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}^{11} & \mathbf{A}^{12} \\ \mathbf{A}^{21} & \mathbf{A}^{22} \end{pmatrix}.$$

- Then e.g., $\mathbf{C} = \mathbf{AB}$ becomes

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

- This blocking can be any number, not just 2×2. Very valuable in computer implementations (often hidden from users).

Vector and matrix operations

- The BLAS: Basic Linear Algebra Subprograms.
- Fortran source, but often optimized in vendor libraries.
- BLAS1: vector operations that go as n. Dot product, multiplication by a scalar, addition...
- BLAS2: matrix/vector operations that go as n^2, like matrix/vector multiplication, or matrix addition.
- BLAS3: matrix/matrix operations that go as n^3, like matrix multiplication.
- *Kernels* to build more complicated linear algebra operations.

Keep in mind...

- Floating-point (FP) arithmetic is *finite-precision*.
- Read the IEEE standards if you're interested...
- Note that for 64-bit floating point the processor is required to work in at least 128 bits, and return initially and 80-bit result (all transparent to the user).

Keep in mind...

- Floating-point (FP) arithmetic is *finite-precision*.
- Read the IEEE standards if you're interested...
- Note that for 64-bit floating point the processor is required to work in at least 128 bits, and return initially and 80-bit result (all transparent to the user).
- If you feel you need quad-precision (128-bit FP), your algorithm is probably a bad approach...

Keep in mind...

- Floating-point (FP) arithmetic is *finite-precision*.
- Read the IEEE standards if you're interested...
- Note that for 64-bit floating point the processor is required to work in at least 128 bits, and return initially and 80-bit result (all transparent to the user).
- If you feel you need quad-precision (128-bit FP), your algorithm is probably a bad approach...
- FP arithmetic is not associative! This can impact exploiting parallelism — don't get exactly the same results.

Machine language

- Brutal, lowest-level programming. Typing instructions into the machine like

  34612

- Meaning: add the two floating-point numbers in registers 1 and 2 and put the result in register 6.

Machine language

- Brutal, lowest-level programming. Typing instructions into the machine like

  34612

- Meaning: add the two floating-point numbers in registers 1 and 2 and put the result in register 6.
- Note that you already have had to make sure the desired operands were in registers 1 and 2!

Machine language

- Brutal, lowest-level programming. Typing instructions into the machine like

  34612

- Meaning: add the two floating-point numbers in registers 1 and 2 and put the result in register 6.
- Note that you already have had to make sure the desired operands were in registers 1 and 2!
- And that you know what to do with the result in register 6. . .

Assembly language

Assembly language

- *Much* more sophisticated. . . . . .

Assembly language

- *Much* more sophisticated. . . . . Instead of
  34612
  you could type
  FX6 X1 + X2

Assembly language

- *Much* more sophisticated. . . . . Instead of
  34612
  you could type
  FX6 X1 + X2
  . . .

Assembly language

- *Much* more sophisticated. . . . . Instead of
  34612
  you could type
  FX6 X1 + X2
  . . .
- This was nothing more than a neater "shorthand".

Assembly language

- *Much* more sophisticated. . . . . Instead of
  34612
  you could type
  FX6 X1 + X2
  . . .
- This was nothing more than a neater "shorthand".
- Still required you understood (completely!) the structure of the processor/machine.

Assembly language

- *Much* more sophisticated. . . . . Instead of
  34612
  you could type
  FX6 X1 + X2
  . . .
- This was nothing more than a neater "shorthand".
- Still required you understood (completely!) the structure of the processor/machine.
- Simply not credible for a larger user community.

Higher-level languages

- What we have today (Fortran, C/C++, etc.). Possible to write in a more natural "human" way.

Higher-level languages

- What we have today (Fortran, C/C++, etc.). Possible to write in a more natural "human" way.
- Continual development and incorporation of more advanced features.

Higher-level languages

- What we have today (Fortran, C/C++, etc.). Possible to write in a more natural "human" way.
- Continual development and incorporation of more advanced features.
- Still requires some understanding of target machine.

Higher-level languages

- What we have today (Fortran, C/C++, etc.). Possible to write in a more natural "human" way.
- Continual development and incorporation of more advanced features.
- Still requires some understanding of target machine.
- Interpreted languages: preeminently Python. Easy and good for prototyping.

Lower-level tasks

- Use libraries! Ignore all comments about "hand-tuned" or "hand-optimized" code. The vendors have already done it!

Lower-level tasks

- Use libraries! Ignore all comments about "hand-tuned" or "hand-optimized" code. The vendors have already done it!
- MKL library from Intel (AMD's ACML no longer supported), nVIDIA's CUDA libraries.

Lower-level tasks

- Use libraries! Ignore all comments about "hand-tuned" or "hand-optimized" code. The vendors have already done it!
- MKL library from Intel (AMD's ACML no longer supported), nVIDIA's CUDA libraries.
- Vendor LAPACK, or ATLAS for e.g., ARM.

Lower-level tasks

- Use libraries! Ignore all comments about "hand-tuned" or "hand-optimized" code. The vendors have already done it!
- MKL library from Intel (AMD's ACML no longer supported), nVIDIA's CUDA libraries.
- Vendor LAPACK, or ATLAS for e.g., ARM.
- In general, don't reinvent, exploit!

Lower-level tasks

- Stick with standards, and vendor-written/supported, or large user base-written/supported languages and libraries.

Lower-level tasks

- Stick with standards, and vendor-written/supported, or large user base-written/supported languages and libraries.
- Loop-level parallelism: OpenMP.

Lower-level tasks

- Stick with standards, and vendor-written/supported, or large user base-written/supported languages and libraries.
- Loop-level parallelism: OpenMP.
- GPU parallelism: OpenACC.

Lower-level tasks

- Stick with standards, and vendor-written/supported, or large user base-written/supported languages and libraries.
- Loop-level parallelism: OpenMP.
- GPU parallelism: OpenACC.
- Coarse-grained parallelism: MPI, or SCALAPACK/BLACS.

Lower-level tasks

- Stick with standards, and vendor-written/supported, or large user base-written/supported languages and libraries.
- Loop-level parallelism: OpenMP.
- GPU parallelism: OpenACC.
- Coarse-grained parallelism: MPI, or SCALAPACK/BLACS.
- Avoid approaches that depend on a few authors, or one group, or that are supported only by a single vendor. Long-term risks are too great.

Lower-level tasks

- Stick with standards, and vendor-written/supported, or large user base-written/supported languages and libraries.
- Loop-level parallelism: OpenMP.
- GPU parallelism: OpenACC.
- Coarse-grained parallelism: MPI, or SCALAPACK/BLACS.
- Avoid approaches that depend on a few authors, or one group, or that are supported only by a single vendor. Long-term risks are too great.
- Hyperthreading requires care.