Basic C/C++
For Science and Engineering
Computer Languages

Higher

Interpreted
- Python
- Ruby
- Java
- Matlab

Compiled
- C++
- FORTRAN
- C
- Pascal

High-Level Language
Assembly Language
Machine Language
Hardware

Push eax; mov eax, C3
FF 09 90 90 24 09

for (int i = 0; i < max; i++) ...

x = A\b (solves system of linear equations)

std::vector<double> myVector;

Productivity

Performance

https://www.webopedia.com/TERM/H/high_level_language.html
We use **low-level** languages (C/C++) for performance-critical engineering and scientific codes and **high-level** languages (Python) for other feature-intensive applications.

Vector: An crucial C++ Data Structure

Vectors are the basis for most, if not all, scientific and engineering codes. A vector represents a contiguous array of elements, typically of Double-Precision Floating Point type:

myVector =

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>2.1</td>
<td>1.4</td>
<td>6.6</td>
<td>0.59</td>
<td>0.0</td>
<td>24.3</td>
</tr>
</tbody>
</table>

Address = Base 0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38

Size = 8 elements
Size (bytes) = 8 elements * bytes/element = 64 bytes

In C++, we can use the Standard Template Library (STL) to define and access vectors:

```cpp
#include <vector>

int main(int argc, char* argv)
{
    std::vector<double> myVector(8);
    myVector[0] = 3.6;
    myVector[1] = 2.1;
    myVector[2] = 1.4;
    myVector[3] = 6.6;
    ...
}
```

Alternatively...

```cpp
#include <vector>

int main(int argc, char* argv)
{
    std::vector<double> myVector;
    myVector.push_back(3.6);
    myVector.push_back(2.1);
    myVector.push_back(1.4);
    myVector.push_back(6.6);
    ...
}
```

Practicum I: Adding Two Vectors
Code: p1.cpp

\[ A = \begin{bmatrix} 3 & 6 & 2 & 0 & -2 & \ldots \end{bmatrix} \]
\[ + \]
\[ B = \begin{bmatrix} 2 & 3 & 1 & 1 & 2 & \ldots \end{bmatrix} \]
\[ = \]
\[ C = \begin{bmatrix} 5 & 9 & 3 & 1 & 0 \ldots \end{bmatrix} \]

https://www.eriksmistad.no/getting-started-with-openccl-and-gpu-computing/
Matrices: Just another vector

Suppose we have a M rows and N columns matrix. We can store its entries in a contiguous vector of size MxN such that:
- Its rows are stored contiguously (row-wise)
- Its columns are stored contiguously (column-wise)

std::vector<double> myMatrix(MxN) =

http://www.guideforschool.com/625348-memory-address-calculation-in-an-array/
Practicum II: Calculate Row & Column Sums
Code: p2.cpp

https://www.geeksforgeeks.org/program-to-find-the-sum-of-each-row-and-each-column-of-a-matrix/
Common Pattern I: Matrix-Vector Multiplication

We have an MxN matrix A and a N-vector b, we want to calculate Ab.

\[
A = \begin{pmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{pmatrix}, \quad b = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad Ab = \begin{pmatrix}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3
\end{pmatrix}
\]

Data Complexity: MN + 2N = O(MN) -> O(N^2) for square matrices.

Procedure:

First row,

\[
\begin{pmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & 4 & 2 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \cdot 3 + 1 \cdot 1 + 1 \cdot 2 \end{pmatrix}
\]

next row,

\[
\begin{pmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & 4 & 2 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \cdot 3 + 1 \cdot 1 + 3 \cdot 2 \end{pmatrix}
\]

last row,

\[
\begin{pmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & 4 & 2 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \cdot 3 + 1 \cdot 1 + 1 \cdot 2 \\ 2 \cdot 3 + 1 \cdot 1 + 3 \cdot 2 \\ 1 \cdot 3 + 4 \cdot 1 + 2 \cdot 2 \end{pmatrix} = \begin{pmatrix} 6 \\ 13 \\ 11 \end{pmatrix}
\]

Computational Complexity: M*N = O(MN) -> O(N^2) for square matrices.
Practicum III: Calculate $Axb$

Code: p3.cpp

\[
\begin{pmatrix}
1 & 1 & 2 \\
2 & 1 & 3 \\
1 & 4 & 2 \\
\end{pmatrix}
\begin{pmatrix}
3 \\
1 \\
2 \\
\end{pmatrix} = \begin{pmatrix}
1 \cdot 3 + 1 \cdot 1 + 1 \cdot 2 \\
2 \cdot 3 + 1 \cdot 1 + 3 \cdot 2 \\
1 \cdot 3 + 4 \cdot 1 + 2 \cdot 2 \\
\end{pmatrix}
= \begin{pmatrix}
6 \\
13 \\
11 \\
\end{pmatrix}
\]

First row,

next row,

last row,
Common Pattern II: Matrix-Matrix Multiplication

We have an MxN matrix A and an NxK, we want to calculate C = AB.

\[
\begin{bmatrix}
    a_{11} & a_{12} & a_{13} \\
    a_{21} & a_{22} & a_{23} \\
    a_{31} & a_{32} & a_{33}
\end{bmatrix}
\begin{bmatrix}
    b_{11} & b_{12} \\
    b_{21} & b_{22} \\
    b_{31} & b_{32}
\end{bmatrix}
= \begin{bmatrix}
    a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\
    a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \\
    a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32}
\end{bmatrix}
\]

Data Complexity: N(K+M). For square matrices: K = M = N -> N(N+N) = 2N^2 -> O(N^2)

Procedure:

Computational Complexity: N*K*M = O(NKM) -> O(N^3) for square matrices.
Practicum IV: Calculate $C = A \times B$
Code: p4.cpp

https://en.wikipedia.org/wiki/Matrix_multiplication
http://xaktly.com/MatrixOperations.html
Cache Hierarchies: A brief History
The Intel 8086 Processor

1978 - Intel Releases 8086, the first 16-bit processor of the x86 architecture.

- 29k Transistors - 5Mhz

16-bit = 64K Max Memory Address

Address Calculator Unit

Segment Selectors (16-bit)

General-Purpose Registers (16-bit)

Further Read: Computer Organization and Design: The Hardware/Software Interface

Picture: https://commons.wikimedia.org/wiki/File:Intel_8086_block_scheme.svg
(32-bit) Extended Registers

1986 - Intel Releases 80386, its first 32-bit Processor.

275k Transistors - 20Mhz

Extended Registers

Applications Operate on Larger Data Sets

32-bit = 4gb Max Memory Address

Additional Pressure on RAM
CPU/RAM Latency

275k Transistors - 20Mhz

GP Register Capacity: 16 Bytes
Register Memory Latency: 2-5ns

Latency Ratio: 24x
Capacity Ratio: ~10^6

Asynchronous DRAM
Capacity: 1-64 Mbytes
Latency: ~120ns

Picture Source: CPU collection Konstantin Lanzet
Memory Performance Gap

Problem - Growing disparity between register and RAM latencies.
Cache Memory

1988 - Intel Releases 80386SX, the first commercial CPU with a Data-Cache Memory

Latency Ratio: 4x
Capacity Ratio: $\sim 10^{4}$

Latency Ratio: 6x
Capacity Ratio: $\sim 10^{2}$

- Cache Memories do not improve the RAM->CPU Latency.
- Instead, they speed-up the reuse of data based on temporal and cache locality.

275k Transistors - 20Mhz
GP Register Capacity: 16 Bytes
Register Memory Latency: 2-5ns

External Cache (SRAM)
Capacity: $\sim 128$ kbytes
Latency: 10-25ns

Asynchronous DRAM
Capacity: 1-64 Mbytes
Latency: $\sim 120$ns

RAM is itself a cache for HDD!
Cache Locality

Cache structures in modern processors benefit from both **Temporal** and **Spatial** locality.

**High Cache Line Reuse**

**Frequent Cache Fails**
Cache Hierarchy

**201x - Modern Processors Employ Multiple Cache Levels**

**Figure 2.1:** Configuration of a 32-core NERSC Cori Phase I (Haswell) Node.
RAM Latency

Solution - Cache memories bridges the gap between CPU and RAM performance
Case Study: Matrix-Matrix Multiplication
Cache Usage: Matrix Matrix Multiplication

A and B stored row-major differentiated color. Let’s ignore C accesses for simplicity.
4 Cache Lines of 2 Elements Each

A

B

C

System Memory

Compulsory Misses #

Capacity Misses #
Question:
How many capacity misses will there be in total?
Cache Optimization: Blocked Multiplication

Idea: Let's solve the multiplication in blocks, storing partial solutions to $C_{i,j}$.
Practicum V:

Answer: How many capacity misses there be in total now?

Develop: A cache-efficient blocked matrix-matrix multiplication for the square matrices in p5.cpp