CS601: Software Development for Scientific Computing

Autumn 2024

Week5: Build tool (Make demo), Motifs – Matrix Computations with Dense Matrices

`make` – Recap and Demo

- Minimal build
  - What if only `scprod.cpp` changes?
- Special targets (`.phony`)
  - E.g. explicit request to `clean` executes the associated recipe. What if there is a file named `clean`?
- Organizing into folders
  - Use of variables (built-in (`CXX, CFLAGS`) and automatic (`$@, $^, $<`))

refer to week4_codesamples

Recall Motifs from Week1

Scientific Software - Motifs

noun

1. a decorative image or design, especially a repeated one forming a pattern.
   "the colourful hand-painted motifs which adorn narrowboats"

   Similar: design pattern decoration figure shape logo monogram

2. a dominant or recurring idea in an artistic work.
   "superstition is a recurring motif in the book"

1. Finite State Machines
2. Combinatorial
3. Graph Traversal
4. Structured Grid
5. Dense Matrix
6. Sparse Matrix
7. FFT
8. Dynamic Programming
9. N-Body (/ particle)
10. MapReduce
11. Backtrack / B&B
12. Graphical Models
13. Unstructured Grid

Matrix Algebra and Efficient Computation

- **Pic source: the Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View (2008)**

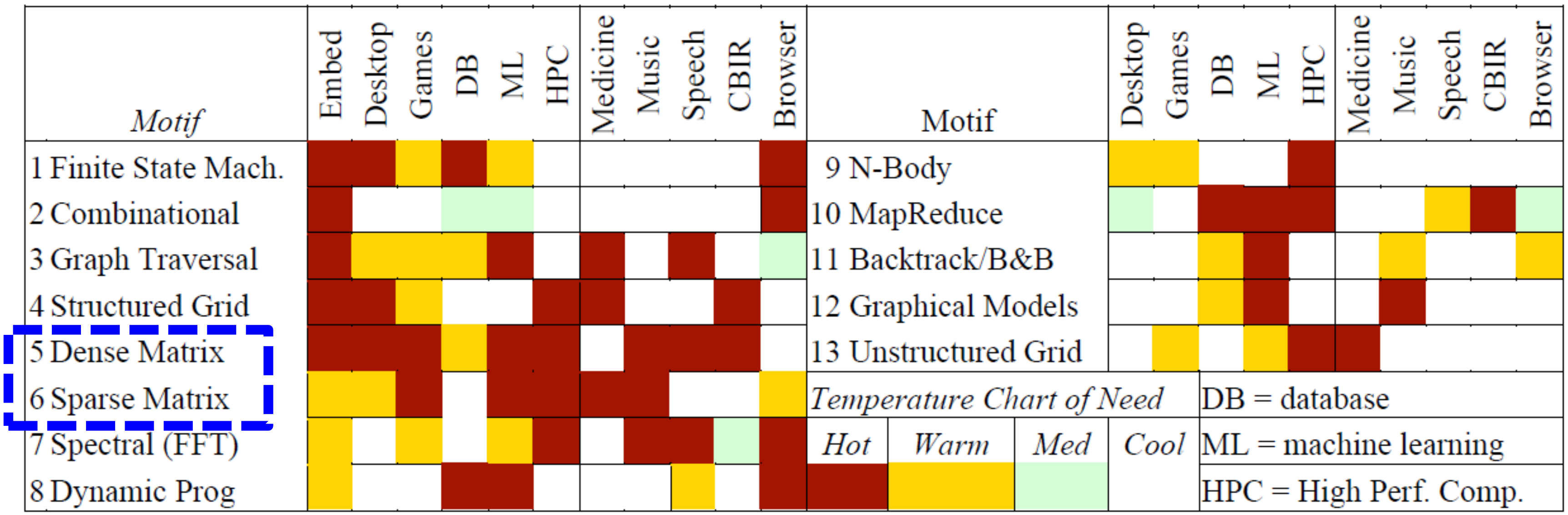


Figure 4. Temperature Chart of the 13 Motifs. It shows their importance to each of the original six application areas and then how important each one is to the five compelling applications of Section 3.1. More details on the motifs can be found in (Asanovic, Bodik et al. 2006).

Matrix Multiplication

- Why study?
  - An important “kernel” in many linear algebra algorithms
  - Most studied kernel in high performance computing
  - Simple. Optimization ideas can be applied to other kernels
- Matrix representation
  - Matrix is a 2D array of elements. Computer memory is inherently linear
  - C++ and Fortran allow for definition of 2D arrays. 2D arrays stored row-wise in C++. Stored column-wise in Fortran. E.g.

```
// stores 10 arrays of 20 doubles each in C++
double** mat = new double[10][20];
```

Storage Layout - Example

- Matrix (**2D**):$A = \begin{bmatrix} A(0,0) & A(0,1) & A(0,2) \\ A(1,0) & A(1,1) & A(1,2) \\ A(2,0) & A(2,1) & A(2,2) \end{bmatrix}$

$A(i,j) = A(row, column)$ refers to the matrix element in the i^{th} row and the j^{th} column

- Row-wise (/Row-major) storage in memory:

$A(0,0)$	$A(0,1)$	$A(0,2)$	$A(1,0)$	$A(1,1)$	$A(1,2)$	$A(2,0)$	$A(2,1)$	$A(2,2)$

- Column-wise (/Column-major) storage in memory:

$A(0,0)$	$A(1,0)$	$A(2,0)$	$A(0,1)$	$A(1,1)$	$A(2,1)$	$A(0,2)$	$A(1,2)$	$A(2,2)$

- **Generalizing data storage order for ND:** last index changes fastest in row-major. Last index changes slowest in col-major.

Storage Layout - Exercise

- For a 3D array (tensor) assume $A(i, j, k) = A(row, column, depth)$

	$A(0,0,0)$	. . .	. . .	. . .	A(2,2,2)
Offset:	0	1	2	. . .	26

- What is the offset of $A(1, 2, 1)$? as per row-major storage?
- What is the offset of $A(1, 2, 1)$? as per col-major storage?

Matrix Multiplication

- Three fundamental ways to think of the computation

Method 1. Dot product

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1.5 + 2.7 & 1.6 + 2.8 \\ 3.5 + 4.7 & 3.6 + 4.8 \end{bmatrix}$$

Method 2. Linear combination of the columns of the left matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \left[5 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 7 \begin{bmatrix} 2 \\ 4 \end{bmatrix} \quad 6 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 8 \begin{bmatrix} 2 \\ 4 \end{bmatrix} \right]$$

Method 3. Sum of outer products

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \left[\begin{bmatrix} 1 \\ 3 \end{bmatrix} \begin{bmatrix} 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 \\ 4 \end{bmatrix} \begin{bmatrix} 7 & 8 \end{bmatrix} \right]$$

Common Computational Patterns

Some patterns that we see while doing Matrix-Matrix product:

1. Dot Product or Inner Product: $x^T y$ ← Method 1
2. Scalar **a** times **x** **p**lus **y**: y=y+ax OR saxpy ← Method 2
   - Scalar times **x**: αx
3. Matrix times x plus y: y=y+Ax ← Method 1
   - generalized axpy OR gaxpy
4. Outer product: $C=C+xy^T$ ← Method 3
5. Matrix times Matrix plus Matrix
   - GEMM or generalized matrix multiplication

Dot Product

- Vector $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, Vector $y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ $\mathrm{x_i}, \mathrm{y_i} \in \mathbb{R}$
- $x^T = [x_1 \quad x_2 \quad \cdot\cdot \quad x_n]$
- Dot Product or Inner Product: $c = x^T y$
  $\mathrm{x}^{\mathrm{T}} \in \mathbb{R}^{1\times n}, y \in \mathbb{R}^{n\times 1}, c\ is\ scalar$
  $$[x_1 \quad x_2 \quad \cdot\cdot \quad x_n] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = [x_1 y_1 + x_2 y_2 + .. + x_n y_n]$$
- E.g. $[1 \quad 2 \quad 3] \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = [1 \times 4 + 2 \times 5 + 3 \times 6] = 32$

AXPY

- Computing the more common (a times x plus y): $y = y + ax$
- $\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} y \\ y_2 \\ \vdots \\ y_n \end{bmatrix} + a \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$

```
..
for i=1 to n
   y[i] = y[i] +  a*x[i]
..
```

- Cost? n multiplications and n additions = **2n** or **O(n)**

Matrix Vector Product

- Computing Matrix-Vector product: $c = c + Ax,\ A \in \mathbb{R}^{m\times r}, x \in \mathbb{R}^{r\times 1}$

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} + \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1r} \\ a_{21} & a_{22} & \cdots & a_{2r} \\ & & \vdots & \\ a_{m1} & a_{m2} & \cdots & a_{mr} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_r \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} + \begin{bmatrix} a_{11}x_1 + & a_{12}x_2 + & \cdots & +a_{1r}x_r \\ a_{21}x_1 + & a_{22}x_2 + & \cdots & +a_{2r}x_r \\ & & \vdots & \\ a_{m1}x_1 + & a_{m2}x_2 + & \cdots & +a_{mr}x_r \end{bmatrix}$$

$\downarrow m \ \rightarrow r$ $\qquad\qquad$ $\downarrow m \ \rightarrow 1$

- Rewriting Matrix-Vector product using dot products:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} + \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1r} \\ a_{21} & a_{22} & \cdots & a_{2r} \\ & & \vdots & \\ a_{m1} & a_{m2} & \cdots & a_{mr} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_r \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} + \begin{bmatrix} a_1^T x \\ a_2^T x \\ \vdots \\ a_m^T x \end{bmatrix}$$

- Cost? m rows involving dot products and having the form $c_i = c_i + x^T y$ (Per row cost = 2r (because $a_i, x \in \mathbb{R}^r$), Total cost = **2mr** or **O(mr)**)

Matrix-Matrix Product

- Computing Matrix-Matrix product $C = C + AB, A \in \mathbb{R}^{m \times r}, B \in \mathbb{R}^{r \times n}, C \in \mathbb{R}^{m \times n}$

$$\begin{bmatrix} c_{11} & c_{12} & .. & c_{1n} \\ c_{21} & c_{22} & .. & c_{2n} \\ & \vdots & & \\ c_{m1} & c_{m2} & .. & c_{mn} \end{bmatrix} = \underset{\downarrow m \ \rightarrow n}{\begin{bmatrix} c_{11} & c_{12} & .. & c_{1n} \\ c_{21} & c_{22} & .. & c_{2n} \\ & \vdots & & \\ c_{m1} & c_{m2} & .. & c_{mn} \end{bmatrix}} + \underset{\downarrow m \ \rightarrow r}{\begin{bmatrix} a_{11} & a_{12} & .. & a_{1r} \\ a_{21} & a_{22} & .. & a_{2r} \\ & \vdots & & \\ a_{m1} & a_{m2} & .. & a_{mr} \end{bmatrix}} \underset{\downarrow r \ \rightarrow n}{\begin{bmatrix} b_{11} & b_{12} & .. & b_{1n} \\ b_{21} & b_{22} & .. & b_{2n} \\ & \vdots & & \\ b_{r1} & b_{r2} & .. & b_{rn} \end{bmatrix}}$$

- Consider the AB part first.

$$\begin{bmatrix} a_{11} & a_{12} & .. & a_{1r} \\ a_{21} & a_{22} & .. & a_{2r} \\ & \vdots & & \\ a_{m1} & a_{m2} & .. & a_{mr} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & .. & b_{1n} \\ b_{21} & b_{22} & .. & b_{2n} \\ & \vdots & & \\ b_{r1} & b_{r2} & .. & b_{rn} \end{bmatrix}$$

Matrix-Matrix Product

$$\begin{array}{cc} A & B \\ \begin{bmatrix} a_{11} & a_{12} & .. & a_{1r} \\ a_{21} & a_{22} & .. & a_{2r} \\ & : & & \\ a_{m1} & a_{m2} & .. & a_{mr} \end{bmatrix} & \begin{bmatrix} b_{11} & b_{12} & .. & b_{1n} \\ b_{21} & b_{22} & .. & b_{2n} \\ & : & & \\ b_{r1} & b_{r2} & .. & b_{rn} \end{bmatrix} \end{array}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + .. + a_{1r}b_{r1} & . & . & a_{11}b_{1n} + a_{12}b_{2n} + .. + a_{1r}b_{rn} \\ . & & . & \\ a_{m1}b_{11} + a_{m2}b_{21} + .. + a_{mr}b_{r1} & . & . & a_{m1}b_{1n} + a_{m2}b_{2n} + .. + a_{mr}b_{rn} \end{bmatrix}$$

Notice that:

- subscript on `a` varies from 1 to `m` in a column (i.e. m rows exist)
- subscript on `a` varies from 1 to `r` in a row (i.e. r columns exist)

Suppose that we treat a_i as a vector of size r and there exist m vectors

$$= \begin{bmatrix} a_1^T b_1 & . & . & a_1^T b_n \\ . & . & & \\ a_m^T b_1 & . & . & a_m^T b_n \end{bmatrix}$$

$a_i^T \in \mathbb{R}^{1\times r}, b_j \in \mathbb{R}^{r\times 1}$

i ranges from 1 to m

j ranges from 1 to n

Matrix-Matrix Product using Dot Product Formulation

- Pseudocode - Matrix-Matrix product: $C = C + AB,\ A \in \mathbb{R}^{m \times r}, B \in \mathbb{R}^{r \times n}, C \in \mathbb{R}^{m \times n}$

```
..
for i=1 to m
    for j=1 to n
        //compute updates involving dot products
```

$$c_{ij} = c_{ij} + a_i^T b_j$$

Matrix-Matrix Product using Dot Product Formulation – Data Access

- Pseudocode - Matrix-Matrix product: $C = C + AB,\ A \in \mathbb{R}^{m\times r}, B \in \mathbb{R}^{r\times n}, C \in \mathbb{R}^{m\times n}$

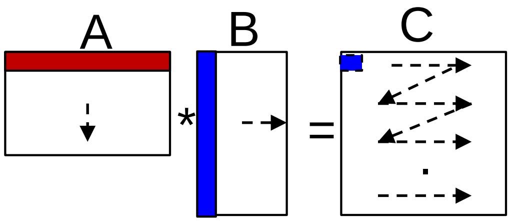
```
..
for i=1 to m
    for j=1 to n
        //compute updates involving dot products
```

$$c_{ij} = c_{ij} + a_i^T b_j$$

- Expanded:

```
..
for i=1 to m
    for j=1 to n
        for k=1 to r
```

$$c_{ij} = c_{ij} + a_{ik} b_{kj}$$



Elements of C matrix are computed from top to bottom, left to right. Per element computation, you need a row of A and a column of B.

Matrix-Matrix Product using Dot Product Formulation - Cost

- Pseudocode - Matrix-Matrix product: $C = C + AB,\ A \in \mathbb{R}^{m\times r}, B \in \mathbb{R}^{r\times n}, C \in \mathbb{R}^{m\times n}$

```
..
for i=1 to m
    for j=1 to n
        //compute updates involving dot products
```

$$c_{ij} = c_{ij} + a_i^T b_j$$

- Cost?
  - Per dot-product cost = `2r` $(a_i, b_j \in \mathbb{R}^r)$ Total cost = **`2mnr`** `or` **`O(mnr)`**

Matrix Multiplication Performance

- Experimental Setup
  - Xeon Gold 6240C processor
  - 2.6GHz clock frequency
  - 2 processor chips
  - 18 cores per chip
  - 2 fused multiply-add units per core (*can do two double-precision floating point ops of multiplication and addition combined per cycle*)
  - *cache subsystem?*

Matrix Multiplication Performance

C=C+A*B, Square matrices, Dimensions = 2048x2048 (`INPUT_SIZE = 2048`)

	Execution Time	**Speedup (w.r.t. Python)**
Python	2088.75s	1.0
C++	92.7s	22.53
+ –O3	41.67s	50.13
+ ikj loop ordering	4.71s	443.47
+ utilizing all cores (parallel)	0.147s	14209.18

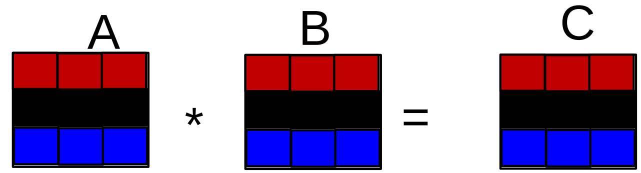
Matrix Multiplication Performance

1. Why ikj loop ordering is fast(er)?
2. Are we utilizing the capabilities of the machine efficiently?

Matrix Multiplication – ikj loop ordering

```
for i=0 to 2
    for k=0 to 2
        for j=0 to 2
```

$$c_{ij} = c_{ij} + a_{ik}b_{kj}$$



When

i,k=0: $C(0,0) += A(0,0) * B(0,0)$ $C(0,1) += A(0,0) * B(0,1)$ $C(0,2) += A(0,0) * B(0,2)$

i=0,k=1: $C(0,0) += A(0,1) * B(1,0)$ $C(0,1) += A(0,1) * B(1,1)$ $C(0,2) += A(0,1) * B(1,2)$

i=0,k=2: $C(0,0) += A(0,2) * B(2,0)$ $C(0,1) += A(0,2) * B(2,1)$ $C(0,2) += A(0,2) * B(2,2)$

i=1,k=0: $C(1,0) += A(1,0) * B(0,0)$ $C(1,1) += A(1,0) * B(0,1)$ $C(1,2) += A(1,0) * B(0,2)$

i=1,k=1: $C(1,0) += A(1,1) * B(1,0)$ $C(1,1) += A(1,1) * B(1,1)$ $C(1,2) += A(1,1) * B(1,2)$

i=1,k=2: $C(1,0) += A(1,2) * B(2,0)$ $C(1,1) += A(1,2) * B(2,1)$ $C(1,2) += A(1,2) * B(2,2)$

i=2,k=0: $C(2,0) += A(2,0) * B(0,0)$ $C(2,1) += A(2,0) * B(0,1)$ $C(2,2) += A(2,0) * B(0,2)$

i=2,k=1: $C(2,0) += A(2,1) * B(1,0)$ $C(2,1) += A(2,1) * B(1,1)$ $C(2,2) += A(2,1) * B(1,2)$

i=2,k=2: $C(2,0) += A(2,2) * B(2,0)$ $C(2,1) += A(2,2) * B(2,1)$ $C(2,2) += A(2,2) * B(2,2)$

Matrix Multiplication – ijk loop ordering

```
for i=0 to 2
    for j=0 to 2
        for k=0 to 2
```

$$c_{ij} = c_{ij} + a_{ik} b_{kj}$$

When

i,j=0: $C(0,0) += A(0,0) * B(0,0)$ $C(0,0) += A(0,1) * B(1,0)$ $C(0,0) += A(0,2) * B(2,0)$

i=0,j=1: $C(0,1) += A(0,0) * B(0,1)$ $C(0,1) += A(0,1) * B(1,1)$ $C(0,1) += A(0,2) * B(2,1)$

i=0,j=2: $C(0,2) += A(0,0) * B(0,2)$ $C(0,2) += A(0,1) * B(1,2)$ $C(0,2) += A(0,2) * B(2,2)$

i=1,j=0: $C(1,0) += A(1,0) * B(0,0)$ $C(1,0) += A(1,1) * B(1,0)$ $C(1,0) += A(1,2) * B(2,0)$

i=1,j=1: $C(1,0) += A(1,0) * B(0,1)$ $C(1,1) += A(1,1) * B(1,1)$ $C(1,1) += A(1,2) * B(2,1)$

i=1,j=2: $C(1,0) += A(1,0) * B(0,2)$ $C(1,2) += A(1,1) * B(1,2)$ $C(1,2) += A(1,2) * B(2,2)$

i=2,j=0: $C(2,0) += A(2,0) * B(0,0)$ $C(2,0) += A(2,1) * B(1,0)$ $C(2,0) += A(2,2) * B(2,0)$

i=2,j=1: $C(2,0) += A(2,0) * B(0,1)$ $C(2,1) += A(2,1) * B(1,1)$ $C(2,1) += A(2,2) * B(2,1)$

i=2,j=2: $C(2,0) += A(2,0) * B(0,2)$ $C(2,2) += A(2,1) * B(1,2)$ $C(2,2) += A(2,2) * B(2,2)$

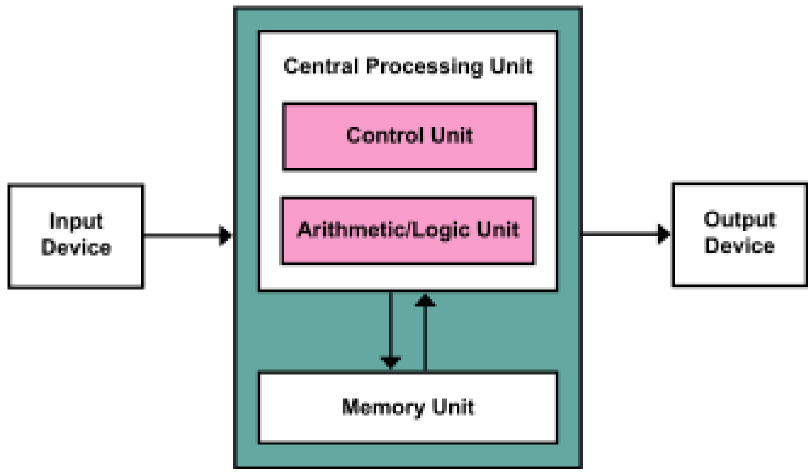
Matrix Multiplication – Data Reuse

- Are we accessing memory location that was read/written recently?
- Are we accessing memory location that is close to one that has been accessed?

Detour – Memory Hierarchy

The von Neumann Architecture

- Proposed by Jon Von Neumann in 1945

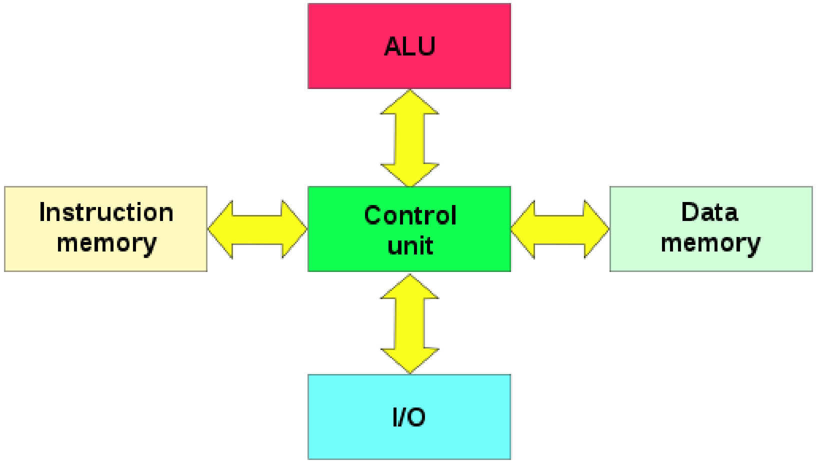


source: wikipedia

- The memory unit stores both instruction and data
  - consequence: cannot fetch instruction and data simultaneously - *von Neumann bottleneck*

Harvard Architecture

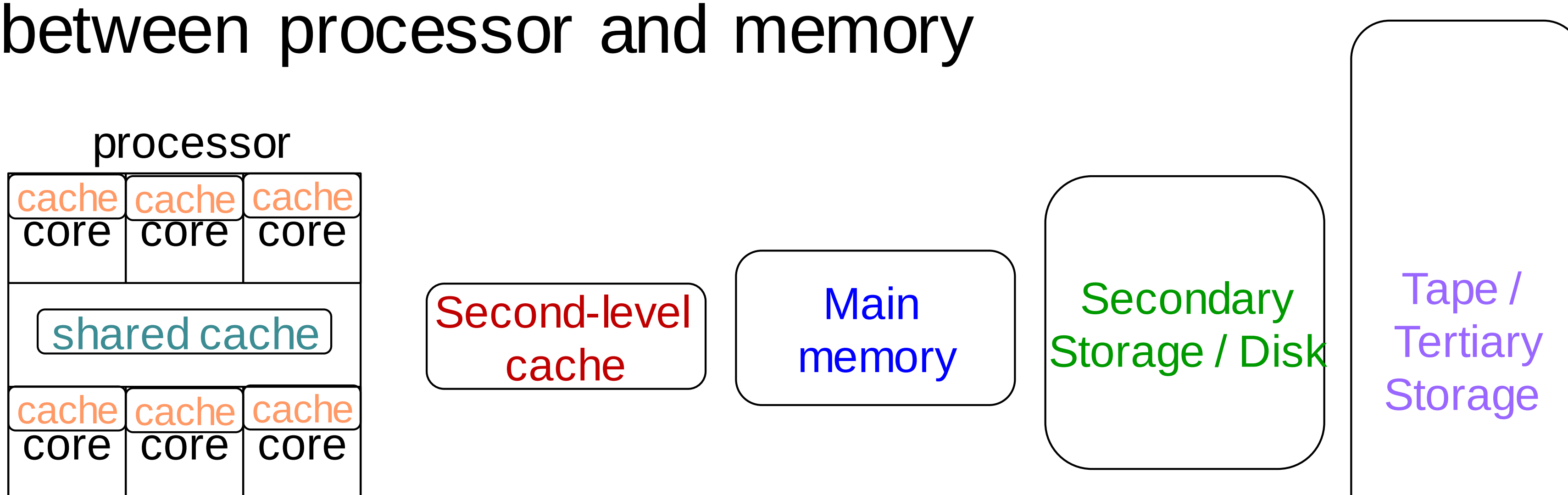
- Origin: Harvard Mark-I machines
- Separate memory for instruction and data



  - advantage: speed of execution
  - disadvantage: complexity

Memory Hierarchy

- Most computers today have layers of cache in between processor and memory



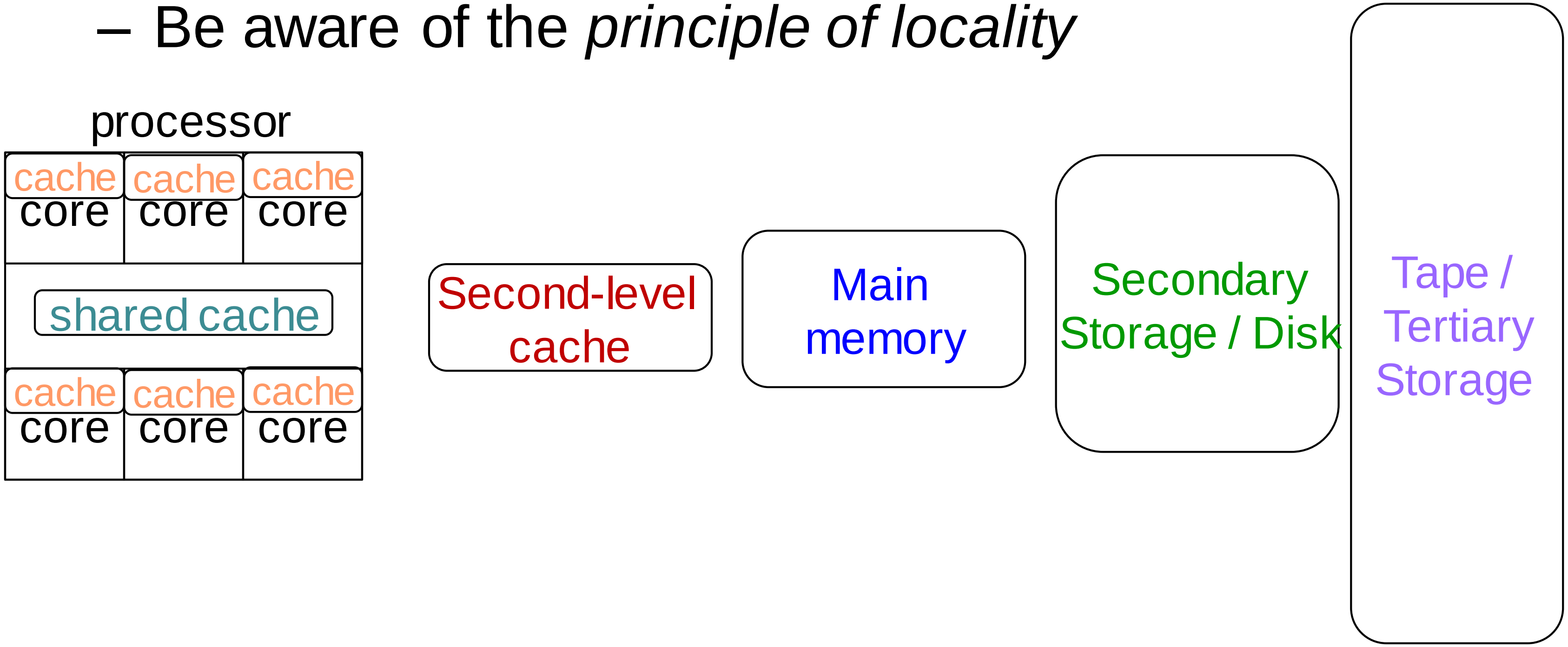
Latency:	1 ns	~5-10 ns	$\sim 10^2$ ns	$\sim 10^7$ ns	$\sim 10^{10}$ ns
Size:	few KBs	$\sim 10^6$ (MBs)	$\sim 10^9$ (GBs)	$\sim 10^{12}$ (TBs)	$\sim 10^{15}$ (PBs)

  – Closer to cores exist separate D and I caches

- Where are *registers?*

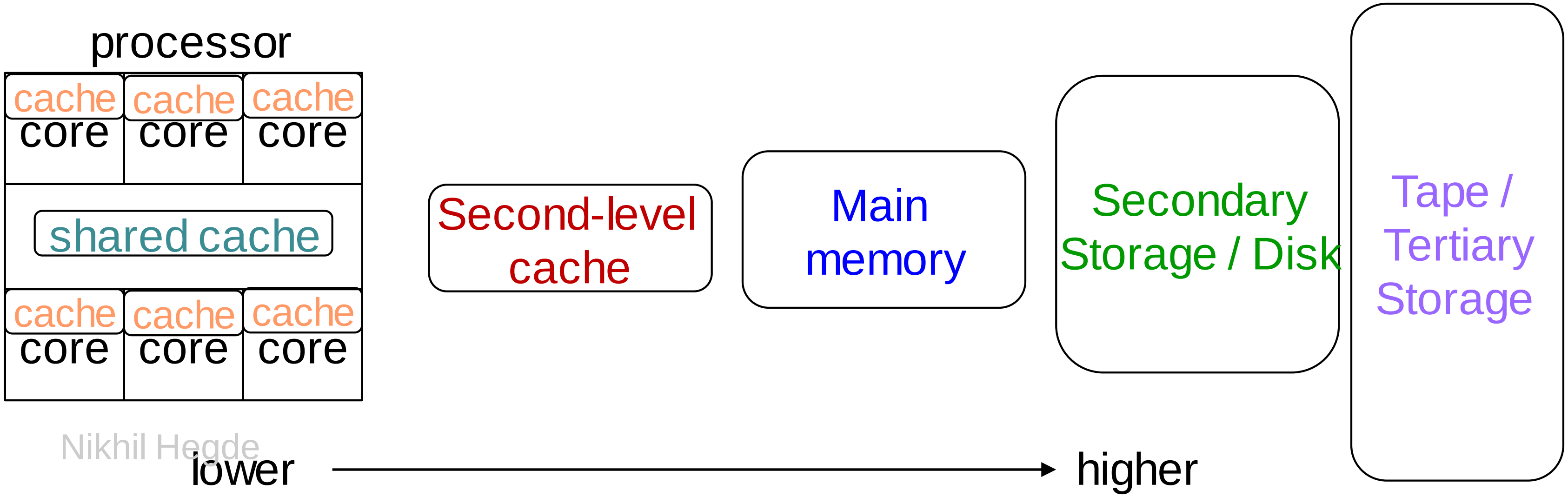
Memory Hierarchy

- Consequences on programming?
  - Data access pattern influences the performance
  - Be aware of the *principle of locality*



Memory Hierarchy - Terminology

- Hit: data found in a lower-level memory module
  - Hit rate: fraction of memory accesses found in lower-level
- Miss: data to be fetched from the next-level (higher) memory module
  - Miss rate: 1 – Hit rate
  - Miss penalty: time to replace the data item at the lower-level



Principle of Locality

1. If a data item is accessed, it will tend to be accessed soon *(temporal locality)*
   – So, keep a copy in cache
   – E.g. loops
2. If a data item is accessed, items in nearby addresses in memory tend to be accessed soon *(spatial locality)*
   – Guess the next data item (based on access history) and fetch it
   – E.g. array access, code without any branching

Demo – Understanding Cache Hierarchy

- How to find the details of cache subsystem on a machine?

  > cat `/sys/devices/system/cpu/cpu0/cache/index0/type`

  tells whether it is either Data / Instruction cache

  – Explore each of the files within to know more.

Matrix Multiplication - Throughput

C=C+A*B, Square matrices, Dimensions = 2048x2048 (`INPUT_SIZE = 2048`)

Peak throughput: $2.6 \times 10^9 \times 2 \times 18 \times 2$ = **187.2 Giga** floating point operations per second (FLOPS)

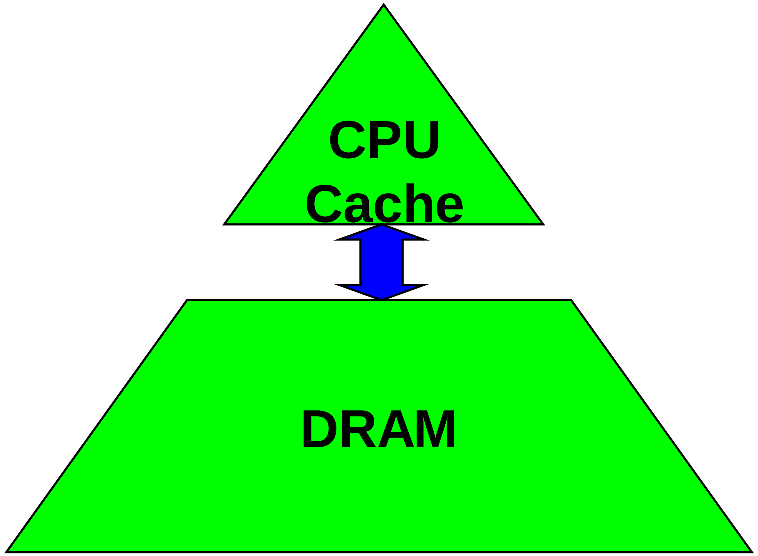
	Execution Time	**Speedup (w.r.t. Python)**	**Throughput (approximate in FLOPS)**
Python	2088.75s	1.0	(2×2^{33}) / 2088.75 = 8.23 **M**
C++	92.7s	22.53	185.33 **M**
+ –O3	41.67s	50.13	412.28 **M**
+ ikj loop ordering	4.71s	443.47	3.65 **G**
+ utilizing all cores (parallel)	0.147s	14209.18	116.87 **G**

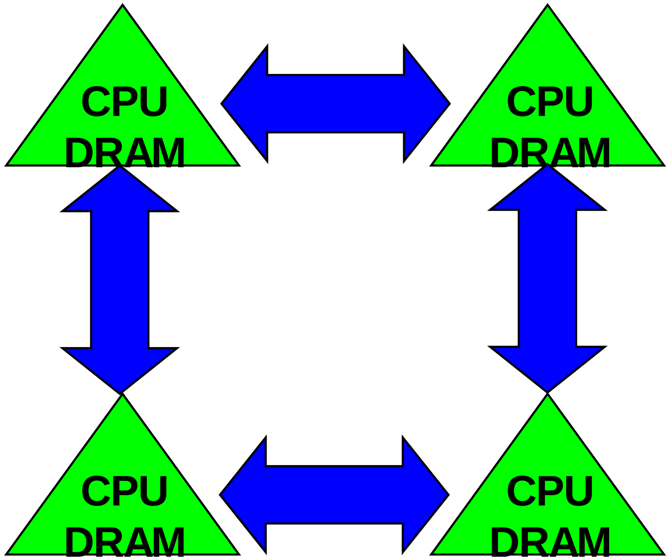
62.3 % of the peak

Costs Involved

Algorithms have two costs:

1. Arithmetic (FLOPS)
2. Communication: moving data between
   – levels of a memory hierarchy (sequential case)
   – processors over a network (parallel case).





Computational Intensity

- Connection between computation and communication cost
- Average number of operations performed per data element (word) read/written from slow memory
  - E.g. Read/written m words from memory. Perform f operations on m words.
  - Computational Intensity q = f/m (*flops per word*).
- Goal: we want to *maximize* the computational intensity
  - We want to minimize words moved (read/written)
  - We want to minimize messages sent

What is the computational intensity, q, for:
axpy?
***M**atrix-**V**ector product?*
***M**atrix-**M**atrix product?*