# CS601: Software Development for Scientific Computing

Autumn 2022

Week4: Build tool (Make contd.), Motifs – Matrix Computations with Dense Matrices

# So far..

- Overview (scientific software, examples, commonly occurring patterns in scientific computing)
- IEEE-754 Representation
- Creating a program (Program Development Environment )

Implementation ⇨ Toolchain ⇨ Executable

- Entry point of execution
- Functions
- Reference variables in C++
- Declaration vs. Definition
- C++ Types (standard, compound)

- Tools that are involved: preprocessor, compiler, assembler, loader, linker

- How to execute?
- How to pass arguments from command line?
- How is the program laid out in memory?

**Towards creating software (`vectorprod_vx.cpp`):**
Data types (flexibility, adaptability)
Correctness (exceptions, validating)
Creating modular code

# Discussion `vectorprod_vx.cpp`

**Refer to:**

- `vectorprod_v1.cpp`
  - What if `atoi` doesn't provide accurate status about the value returned?
- `vectorprod_v2.cpp`
  - C++ stringstreams are an option. Is this code modular?
- `vectorprod_v3.cpp scprod.cpp`
  - What if there is already built-in function by the same name?
- `vectorprod_v4.cpp scprod_v4.cpp`
  - Namespaces

# `Make` - Recap

# `Makefile` or `makefile`

- Is a file, contains instructions for the `make` program to generate a *target* (executable).
- Generating a target involves:
  1. Preprocessing (e.g. strips comments, conditional compilation etc.)
  2. Compiling ( .c -> .s files, .s -> .o files)
  3. Linking (e.g. making printf available)
- A `Makefile` typically contains directives on how to do steps 1, 2, and 3.

# `Makefile` - Format

1. Contains series of 'rules'-

```
target: dependencies
[TAB]  system command(s)
```

*Note that it is important that there be a TAB character before the system command (not spaces).*

Example: "Dependencies or Prerequisite files" "Recipe"

"target file name"

```
testgen: testgen.cpp
    g++ testgen.cpp –o testgen
```

2. And Macro/Variable definitions -

```
CFLAGS = -std=c++11 -g -Wall -Wshadow --pedantic -Wvla –Werror
GCC = g++
```

# Makefile - Usage

– The ‘make’ command (Assumes that a file by name ‘makefile’ or ‘Makefile’. exists)

```
n2021/slides/week4_codesamples$ cat makefile
vectorprod: vectorprod.cpp scprod.cpp scprod.h
        g++ vectorprod.cpp scprod.cpp -o vectorprod
```

- Run the ‘make’ command

```
n2021/slides/week4_codesamples$ make
g++ vectorprod.cpp scprod.cpp -o vectorprod
```

# `Makefile` - Benefits

- Systematic dependency tracking and building for projects
  - Minimal rebuilding of project
  - Rule adding is ‘declarative’ in nature (i.e. more intuitive to read *caveat: make also lets you write equivalent rules that are very concise and non-intuitive.*)
- To know more, please read: https://www.gnu.org/software/make/manual/html_node/index.html#Top

# `make` - 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 week3_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)**

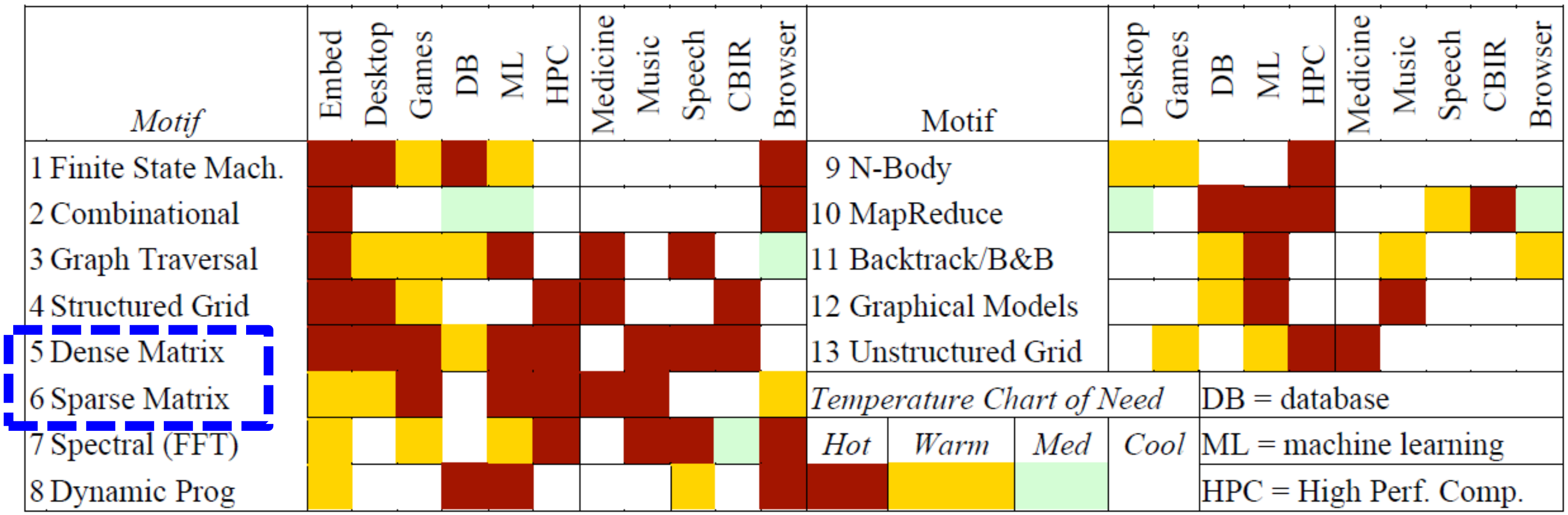



**Figure 4. Temperature Chart of the 13 Motifs.** It shows their importance to each of the original six application areas and then how important each one is to the five compelling applications of Section 3.1. More details on the motifs can be found in (Asanovic, Bodik et al. 2006).

Next..

# 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**):$\mathrm{A} = \begin{bmatrix} A(0,0) & A(0,1) & A(0,2) \\ A(1,0) & A(1,1) & A(1,2) \\ A(2,0) & A(2,1) & A(2,2) \end{bmatrix}$

$A(i,j) = A(row, column)$ 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?

# Storage Layout

- Layout format itself doesn't influence efficiency (i.e. no general answer to "is column-wise layout better than row-wise?" )
- However, knowing the layout format is critical for good performance
  - *Always traverse the data in the order in which it is laid out*

How good performance?

```
Run on (12 X 2592.01 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x6)
  L1 Instruction 32 KiB (x6)
  L2 Unified 256 KiB (x6)
  L3 Unified 12288 KiB (x1)
Load Average: 0.07, 0.02, 0.07
-----------------------------------------------------------------------------------
Benchmark                          Time             CPU   Iterations UserCounters...
-----------------------------------------------------------------------------------
BM_AddByRow/64/64                693 ns          693 ns      1042737 items_per_second=5.91004G/s
BM_AddByRow/128/128             2464 ns         2464 ns       271766 items_per_second=6.64813G/s
BM_AddByRow/256/256            11134 ns        11133 ns        63210 items_per_second=5.88639G/s
BM_AddByRow/512/512            44353 ns        44353 ns        15576 items_per_second=5.91041G/s
BM_AddByCol/64/64               3270 ns         3270 ns       212929 items_per_second=1.25254G/s
BM_AddByCol/128/128            39741 ns        39741 ns        17617 items_per_second=412.272M/s
BM_AddByCol/256/256           314880 ns       314878 ns         2241 items_per_second=208.132M/s
BM_AddByCol/512/512          1276733 ns      1276723 ns          545 items_per_second=205.326M/s
```

Source code: https://github.com/eliben/code-for-blog/tree/master/2015/benchmark-row-col-major

```
des/week13_codesamples$ ./a.out 4096
Rowwise time n=4096 (us): 18967
Colwise time n=4096 (us): 158608
nikhilh@ndhpc01:/mnt/c/temp/Nikhil/Cou
des/week13_codesamples$ ./a.out 2048
Rowwise time n=2048 (us): 4860
Colwise time n=2048 (us): 32158
nikhilh@ndhpc01:/mnt/c/temp/Nikhil/Cou
des/week13_codesamples$ ./a.out 1024
Rowwise time n=1024 (us): 1125
Colwise time n=1024 (us): 1980
```

Matrix-Matrix Addition benchmarking

(Source code and further reading )

Matvec execution time

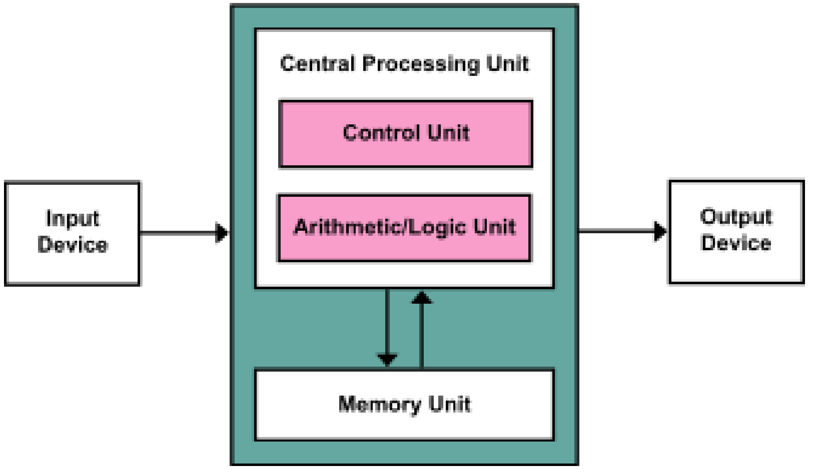
(we used the source code as a basic example to demonstrate row_major vs. col_major storage.)

*refer to week4_codesamples*

# Detour - Memory Hierarchy

# The von Neumann Architecture

- Proposed by Jon Von Neumann in 1945

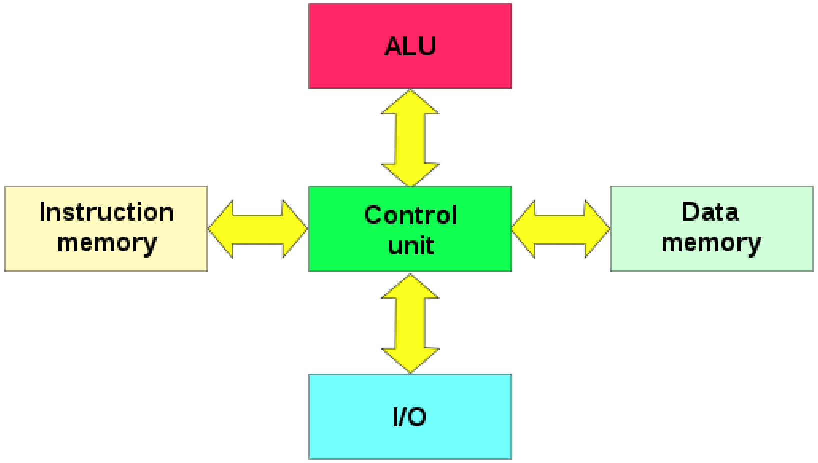



source: wikipedia

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

# Harvard Architecture

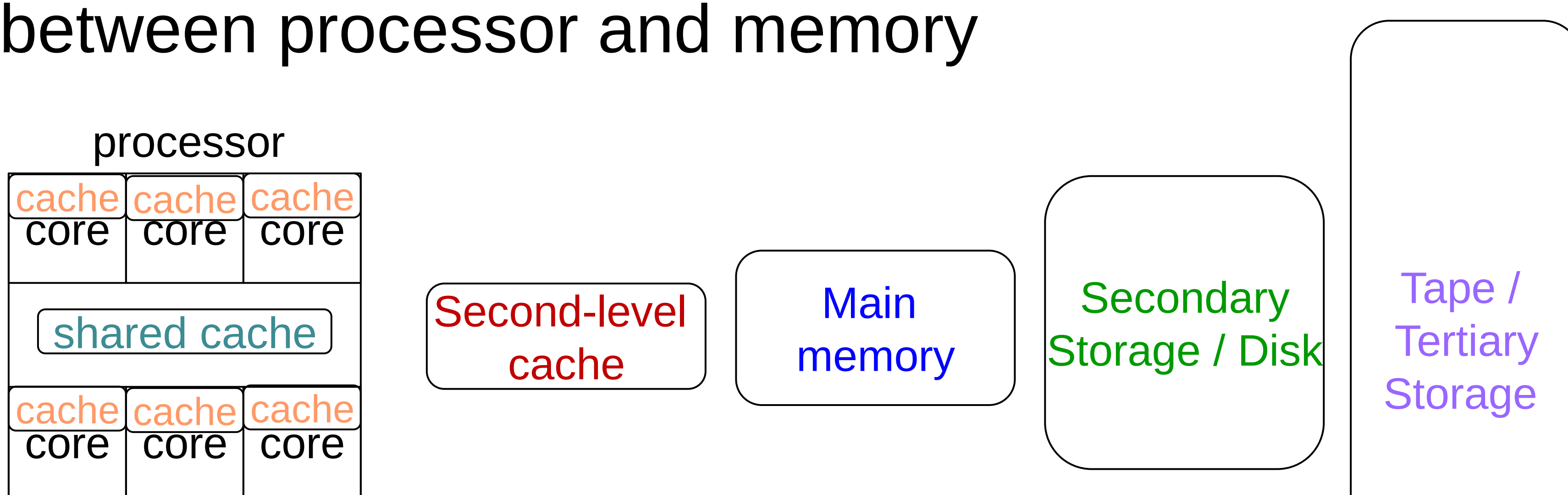
- Origin: Harvard Mark-I machines
- Separate memory for instruction and data




  - advantage: speed of execution
  - disadvantage: complexity

# Memory Hierarchy

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

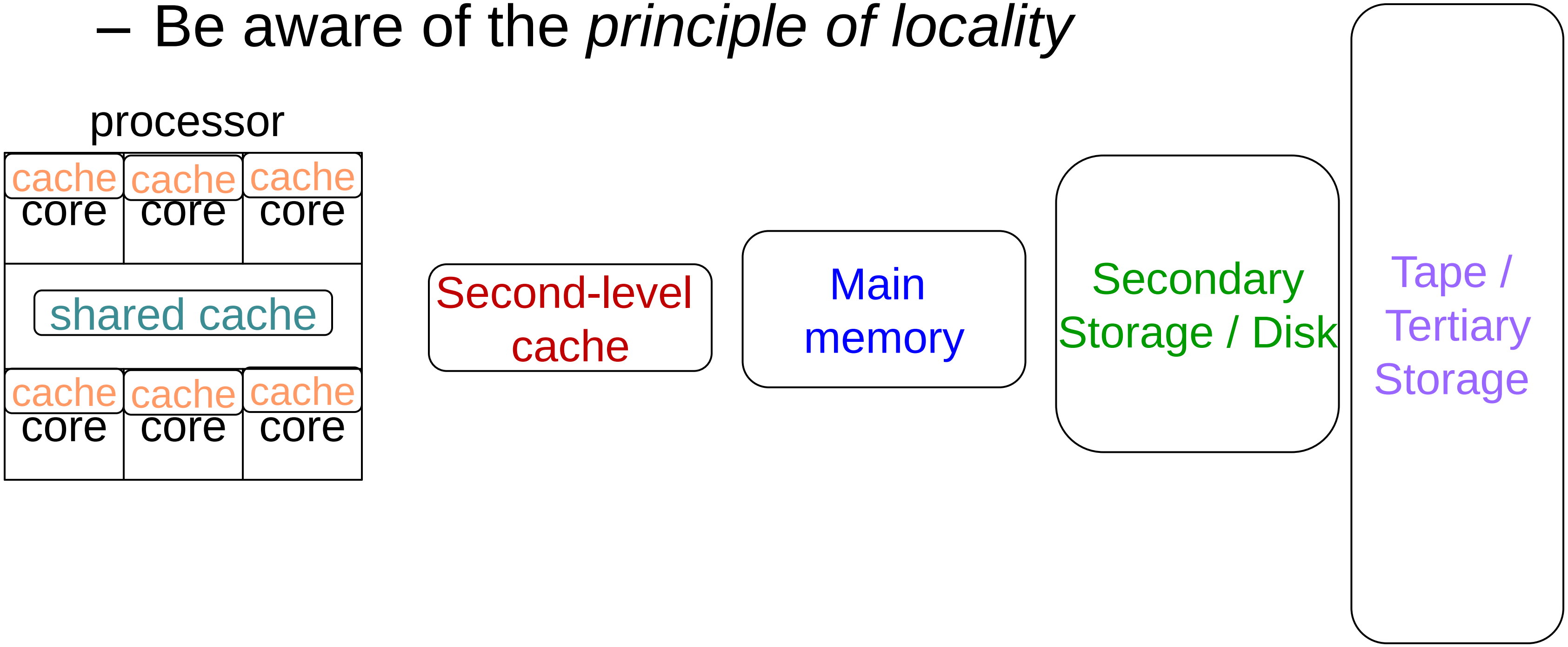



| | | | | | |
|---|---|---|---|---|---|
| **Latency:** | 1 ns | ~5-10 ns | ~$10^2$ ns | ~$10^7$ ns | ~$10^{10}$ ns |
| **Size:** | few KBs | ~$10^6$ (MBs) | ~$10^9$ (GBs) | ~$10^{12}$ (TBs) | ~$10^{15}$ (PBs) |

  - Closer to cores exist separate D and I caches
- Where are *registers?*

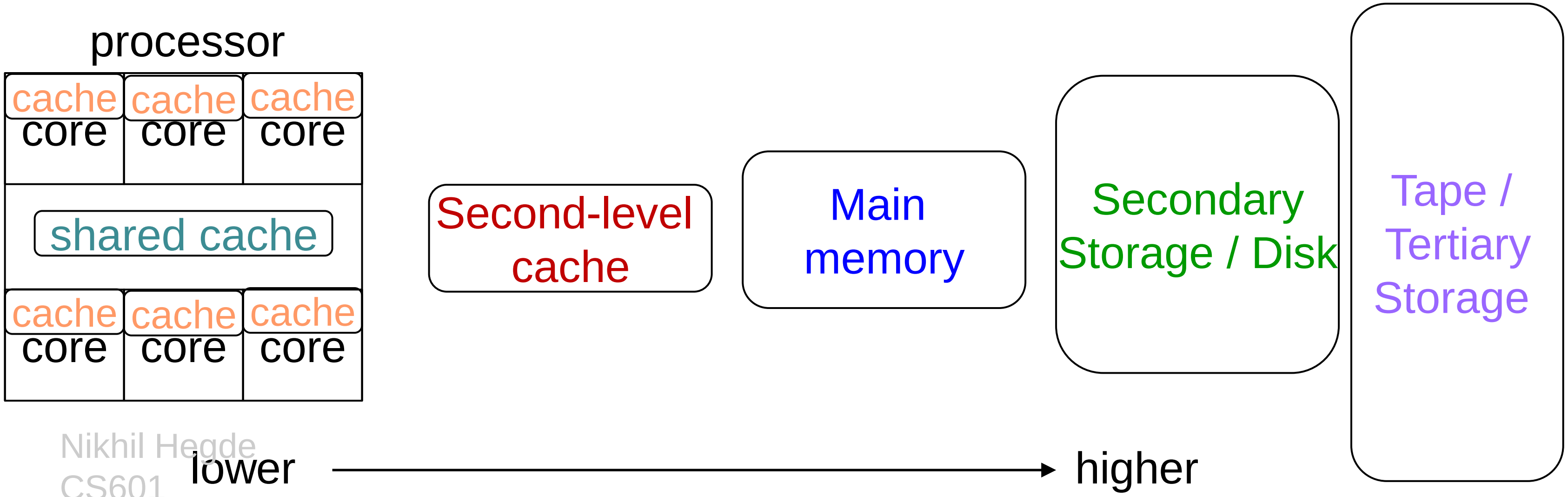
# Memory Hierarchy

- 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.

# Demo – Understanding Performance with PAPI

- PAPI – Performance API
  - Used to count *events* - signals related to processor or other subsystem
  - Processor manufacturers make provision for a small number of registers that count events e.g. floating point operations, cache misses etc.
  - The APIs of PAPI provide a software abstraction to read the platform dependent counters

- refer to `matvec_rowmajor.cpp`, `matvec_colmajor.cpp`, and `makefile` in papi_demo folder of week4_codesamples.
- To build this code using PAPI:
  - you must download [PAPI] and install on your home drive:
    - For installation instructions, read the `INSTALL.txt` file in the downloaded folder.
  - Once installed, you need to change the `CFLAGS` and `LDFLAGS` path in the `makefile`.
  - Now, you can build using `make DEBUG=1` command
  - Before executing the program, on the terminal type: `export LD_LIBRARY_PATH=<absolute-path-where-you-have-installed-papi/lib>`.
  - Now execute using: `./matvec_rowmajor 4096` OR `./matvec_colmajor 4096`.

# Matrix Multiplication

- Three fundamental ways to think of the computation
  1. Dot product

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

  2. Linear combination of the columns of the left matrix

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

  3. Sum of outer products

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

# 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: $c = x^T y$ $\mathrm{x}^\mathrm{T} \in \mathbb{R}^{1\times n}, y \in \mathbb{R}^{n\times 1}, c\ is\ scalar$

$$[x_1 \quad x_2 \quad \cdot\cdot \quad x_n] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = [x_1 y_1 + x_2 y_2 + \ldots + x_n y_n]$$

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

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

- Rewriting Matrix-Vector product using dot products:

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

- 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

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

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

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)

Suggesting that we can 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