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)

- 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.

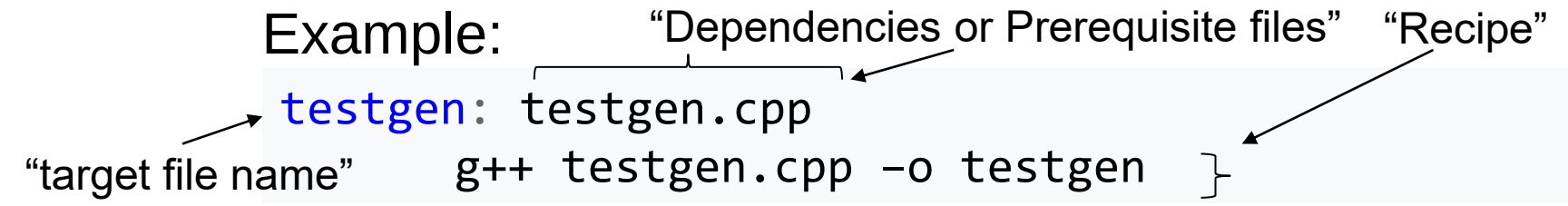
Makefile - Format

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:



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

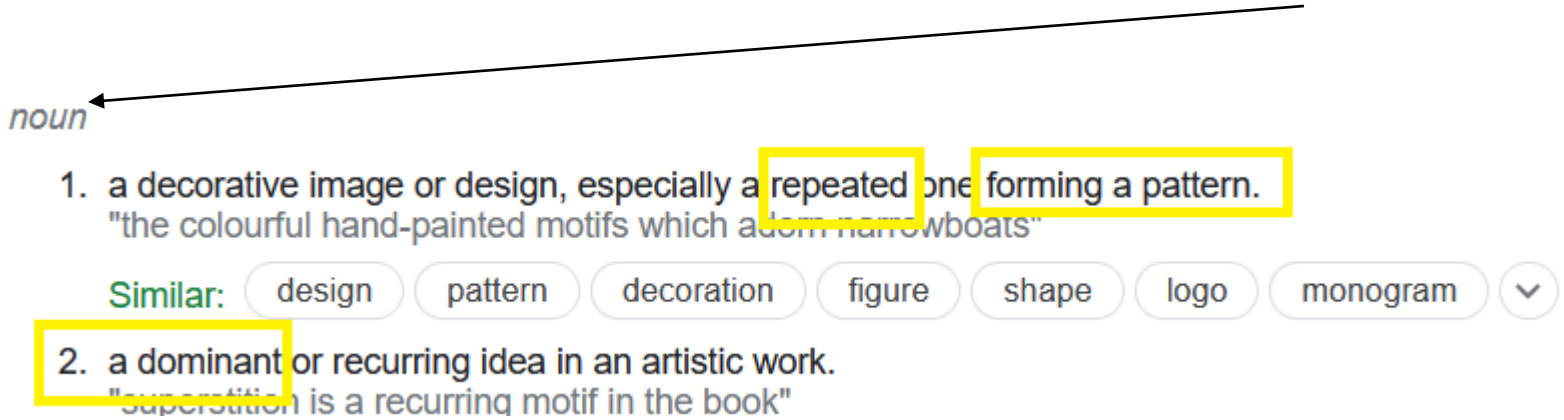
`make` - Demo

- 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



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)**

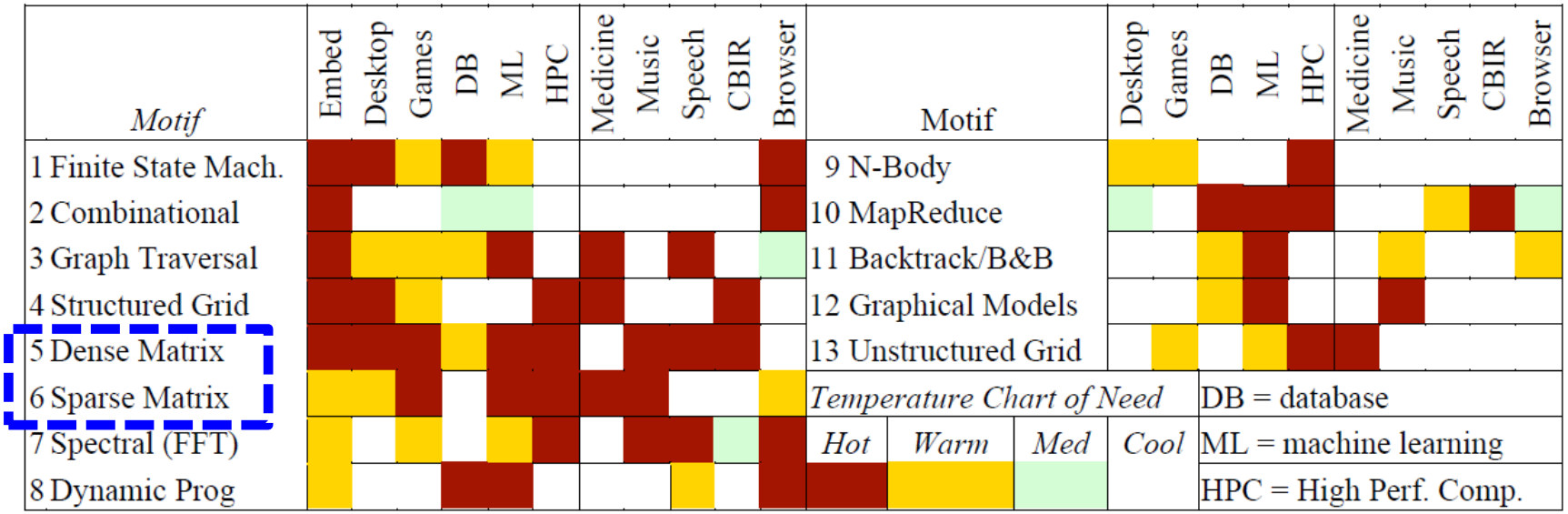


Figure 4. Temperature Chart of the 13 Motifs. It shows their importance to each of the original six application areas and then how important each one is to the five compelling applications of Section 3.1. More details on the motifs can be found in (Asanovic, Bodik et al. 2006).

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

$A(i,j) = A(row, column)$ refers to the matrix element in the ith row and the jth 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 $A(i,j,k) = A(row, column, depth)$

- 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
```

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

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

```
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

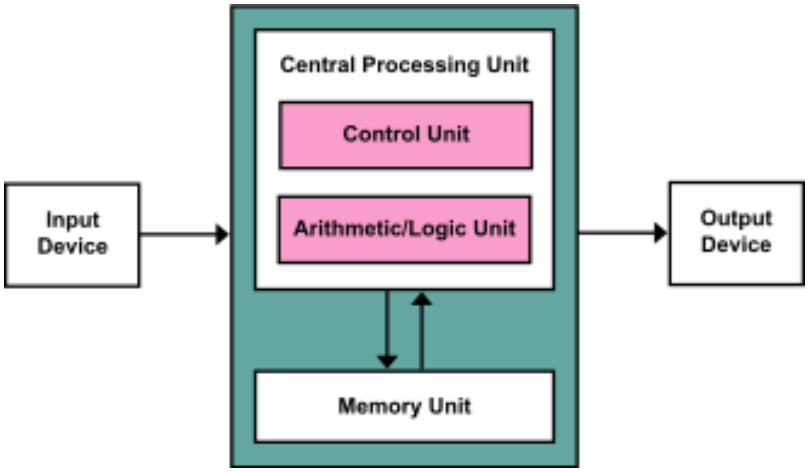
(we used the source code as a basic example to demonstrate row_major vs. col_major storage.)

refer to week4_codesamples

Detour - Memory Hierarchy

The von Neumann Architecture

- Proposed by Jon Von Neumann in 1945

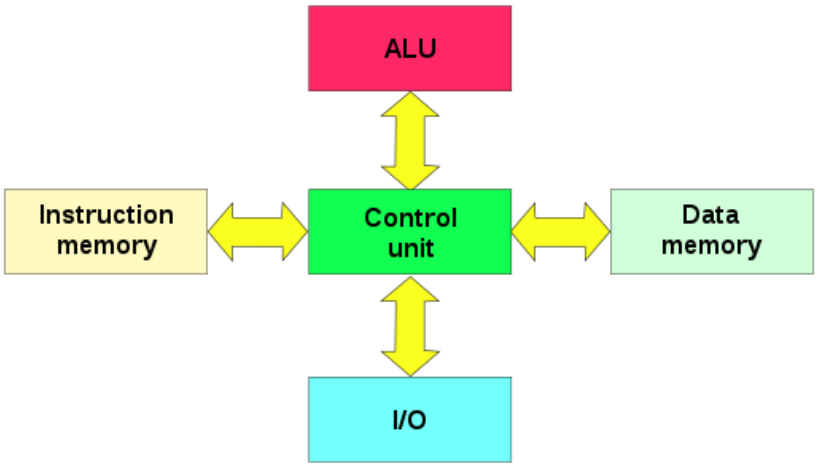


source: wikipedia

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

Harvard Architecture

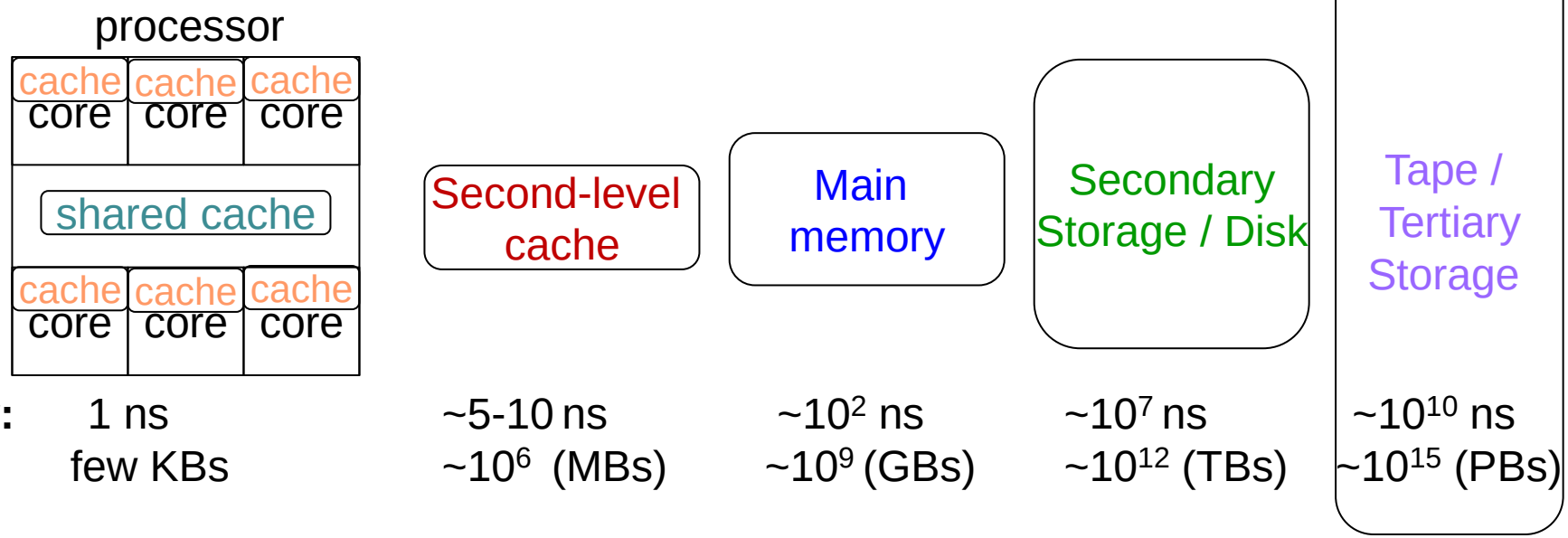
- Origin: Harvard Mark-I machines
- Separate memory for instruction and data



  - advantage: speed of execution
  - disadvantage: complexity

Memory Hierarchy

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



  - 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*