CMOR 421/521:
Arrays and Memory

Section: Serial optimization, Pre-Parallelism

Date: 1/11/2024

T/Th: Overview, review of C/C++

Th: Pointers, Memory, and Architecture

M	W	F	
			1
			2
			3
			4
			5
			6
			7
			8
			9
			10
			11
			12
			13
			14
			15

Topics

- Details of arrays; pointers and memory
- Stack and heap memory
- Caches and memory hierarchy
- A close look at arrays…
  - Multi-dimensional arrays and stride
  - Contiguous vs non-contiguous memory

Why spend so much time on memory?

- Ken Batcher: “A supercomputer is a device for turning compute-bound problems into I/O-bound problems.”
- Theoretical models involving memory are more general than you might expect (e.g., fast/slow memory models or roofline applicable beyond CPUs)
- Getting parallel scalability is often about properly accessing memory in parallel

Recall: Dynamic Mem. Allocation

C++

```
int* x = NULL;
int* array = NULL;

// Allocates a single int and an array of ints
x = new int;
array = new int[<number of array elements>];

// Must free memory!
delete x;
delete[] array;
```

"int* x" is a *pointer* to memory which stores int types.

Pointers: What are they?

Pointers are addresses of locations in memory. They’re represented in hexadecimal (base 16) integers.

- **Base 16?**
  - Aka hex/hexadecimal: “0x2E4C”
  - Decimal is base 10 (what we are familiar with)
  - Binary is base 2, Octal is base 8

Why Hex?

Hexadecimal numbers can encode more information in fewer digits than smaller bases

Decimal (10)	Binary (2)	Hexadecimal (16)
1	0000 0001	1
2	0000 0010	2
3	0000 0011	3
8	0000 1000	8
15	0000 1111	F
16	0001 0000	10
255	1111 1111	FF

Why Represent Pointers in Hex?

- **They can encode more info in fewer digits**
- Modern computers have huge amounts of RAM (gigabyte: 10^9 bytes)
- Bases that are powers of 2 can interface with binary more easily
- Memory is aligned according to powers of 2

What memory is *safe* for pointers?

- **Stack memory** holds *statically* declared variables
  - **"int x[5] = {…}" allocates on the stack**
  - The amount of stack memory needed should be determined at compile time

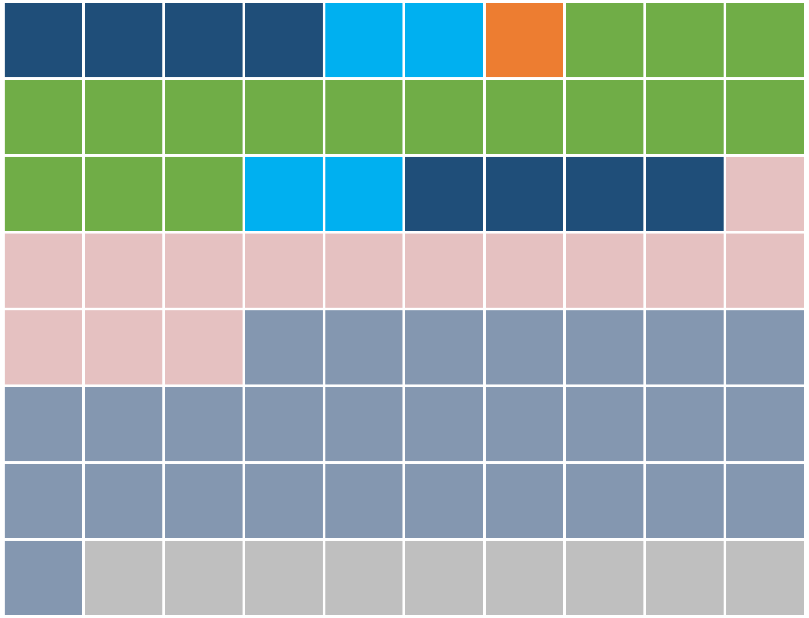
- **Heap memory** is for dynamically allocated variables
  - The amount of heap memory needed is not known until runtime and can vary between runs

- They are handled in different ways by the compiler

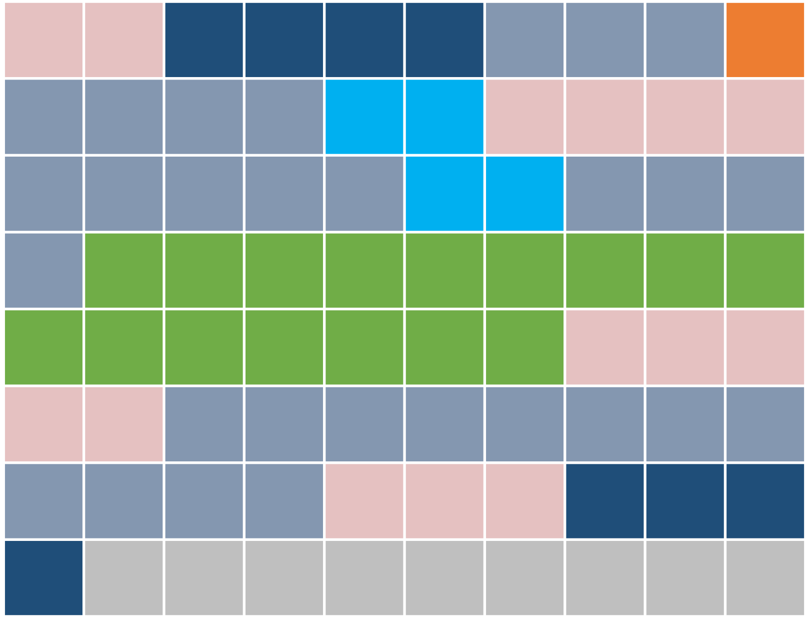
Stack and Heap Management

- **Stack:** Programs/subprograms (functions) are given memory when they are invoked and this memory is freed when they complete
- **Heap:** Programs can allocate and free memory whenever they want throughout their life

Stack



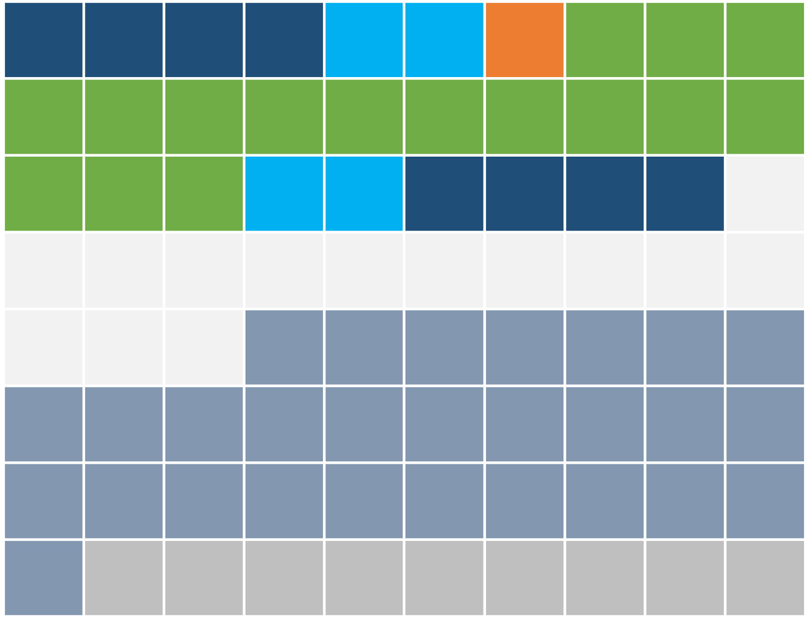
Heap



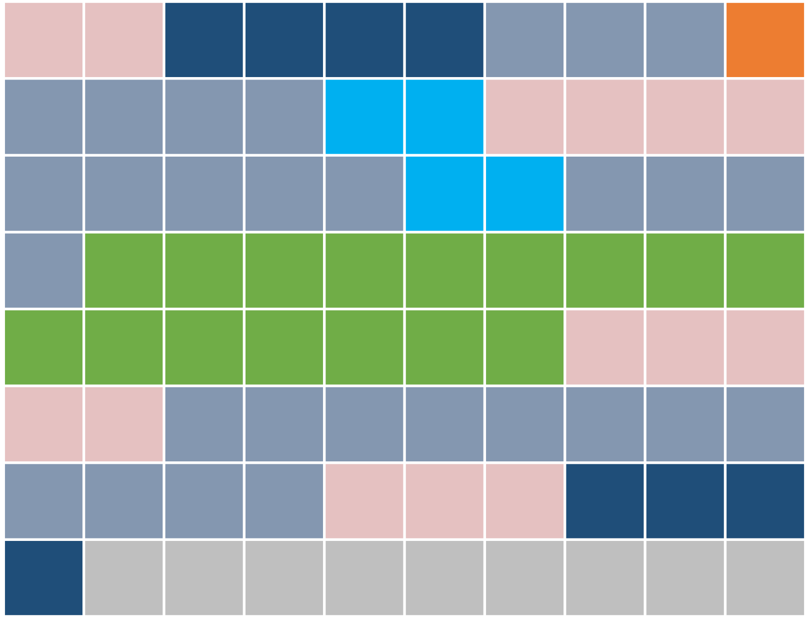
Stack and Heap Management

Stack: When a program finishes, its stack memory is returned to the computer; the computer may rearrange to keep the stack contiguous

Stack

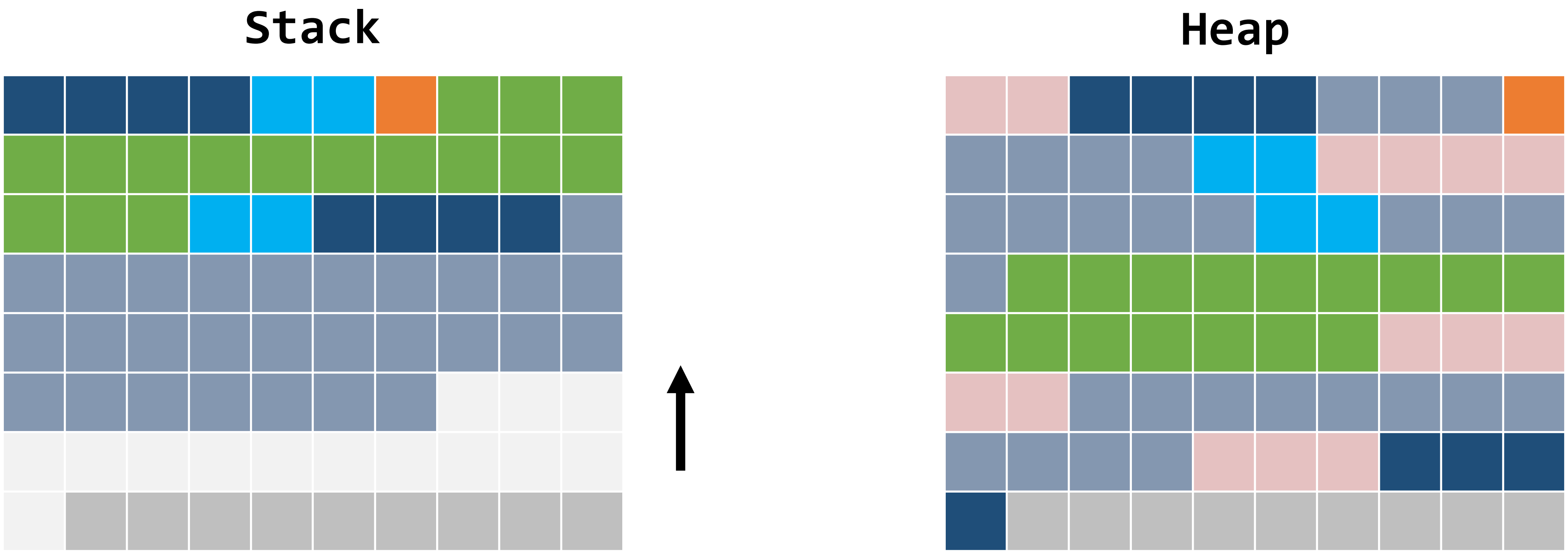


Heap



Stack and Heap Management

- **Stack: Pointers to stack memory are dangerous** because they can become undefined as programs close



Big Takeaways: Stack and Heap

- **Stack:** Relatively small, can overflow and crash
  - **Don’t use pointers to variables in stack memory (e.g., non-dynamically allocated variables)**
  - You won’t get a stack overflow from too many single vars
    - Stack overflow is typically seen with recursion gone wrong, but can also be triggered accidentally in parallel

- **Heap:** Relatively large
  - Can become fragmented which causes allocations to fail and *memory thrashing* (not especially common, but can happen with memory mismanagement in parallel)

There is more still…

- **Computer memory is managed in complex ways**
  - MMU: Memory management unit
  - Virtual memory and physical memory
  - Page tables
  - etc

This is not a course on assembly/computer memory

We will skip these topics

Memory: Bits

00001010100101010001010001010010…

Memory: ~~Bits~~ -> Bytes

00001010 10010101 00010100 01010010…

1 byte = 8 bits (8 binary digits)

Memory: bits and addresses

1
0x00000001

32
0x00000020

00001010 10010101 00010100 01010010…

15
0x0000000f

Memory: Back to Bits

Computers have a lot of memory, and the memory is “aligned” with power of 2

00001010100101010001010001010010…

01001010010100010100000010101000…

01010101001010101010000001001000…

00010100101111001000101001000001…

Arrays, variable types, and pointers organize these bits into interpretable “chunks”.

Why do pointers have types?

- **Recall: there is no type “pointer”.**
- We define “**int** * ptr;” or “**double** * ptr;”
- If we want to access a variable at the address given, the computer needs to know how much to grab

Bytes ->	1	2	3	4	5	6	7	8	9	10	11	12
int (32 bits)												
double (64 bits)												
Struct (int + double)												
int[3]												

What happens when you access an array?

- “int * x = new int [3]”
  - allocates memory for 3 ints
  - binds the pointer “x” to the address of “x[0]”
- What does “x[i]” actually do?
  - accesses the address of “x[0]”
  - steps forward “i * sizeof(int)” bits in memory (e.g., performs pointer arithmetic)
  - returns the value at that memory address.

Bytes ->	1	2	3	4	5	6	7	8	9	10	11	12
int[3]	x[0]				x[1]				x[2]			

Declaring 1D Arrays

C++

```cpp
// Statically defined array
int A[5];

// Dynamically allocated array
int* B = NULL;
B = new int[5];
delete[] B;
```

Multidimensional Arrays

- In computing, we typically need more than 1D arrays
- Examples:
  - 2D: Matrices, grayscale images
  - 2D: Discretized representations of a 2D function
  - 3D: 3D images/data, discretizations of of 3D functions
  - 4D: Time-varying 3D data
  - (*n*xM)D: Representing a system with *n* variables in M spatial dimensions.

Declaring 2D Arrays

C++

```cpp
// Statically defined 2D array
int A[2][4];

// 2D array using an initializer list
int B[][4] = {
   {0, 1, 2, 3},
   {4, 5, 6, 7}
};
// Notice the second dimension must be provided!
```

Multidimensional Arrays in Memory

- **Recall:** pointers need a type so the compiler knows how much memory to grab when access that address
- The type also tells the compiler how big a step to take when indexing an array

Bytes ->	1	2	3	4	5	6	7	8	9	10	11	12
int (32 bits)												
double (64 bits)												
Struct (int + double)												
int[3]												

Arrays and Pointer Arithmetic

- **Pointer arithmetic:** If you increment a pointer, it will increase by increments of its data type
- An array var is a pointer: A[i] = *(A + i)

int[5]	1				2				3				4				5			
Ptr offset	0				+1				+2				+3				+4			

ptr

ptr+1

Pointer arithmetic yields zero-indexing

The first element of an array is A[0], the second A[1], etc

Arrays and Pointer Arithmetic

- Pointer arithmetic means the compiler needs to know the sizes of **all but the first** dimension of mD arrays
- **Notice:** an increment of 1 steps over different amounts of memory for i vs for j. This gives us the notion of ***stride***.

int A[3][4];

i\j	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

A[1][2] = 6

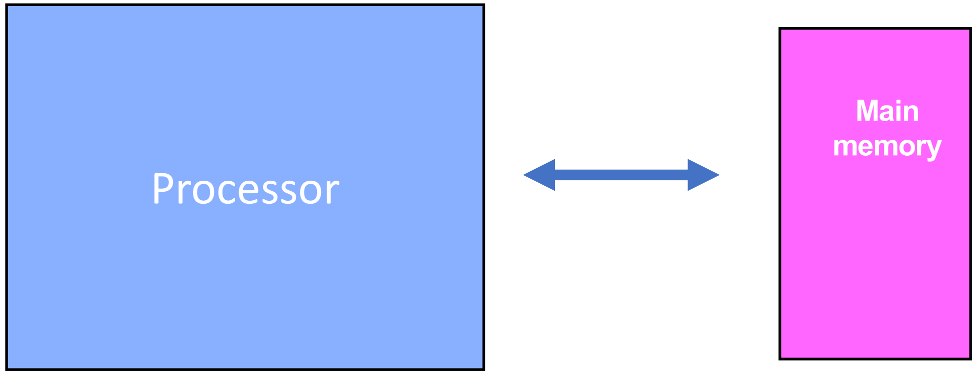
In memory, mD arrays are typically flat arrays

A	0	1	2	3	4	5	6	7	8	9	10	11
A[i][0]	0				1				2			
A[i_const][j]	0	1	2	3	0	1	2	3	0	1	2	3
A[i_flat]	0	1	2	3	4	5	6	7	8	9	10	11

```
1D: A[i]          = *(A + i)
2D: A[i][j]       = *(A + n2*i + j)
    A[0][i_flat] = *(A + i_flat)
```

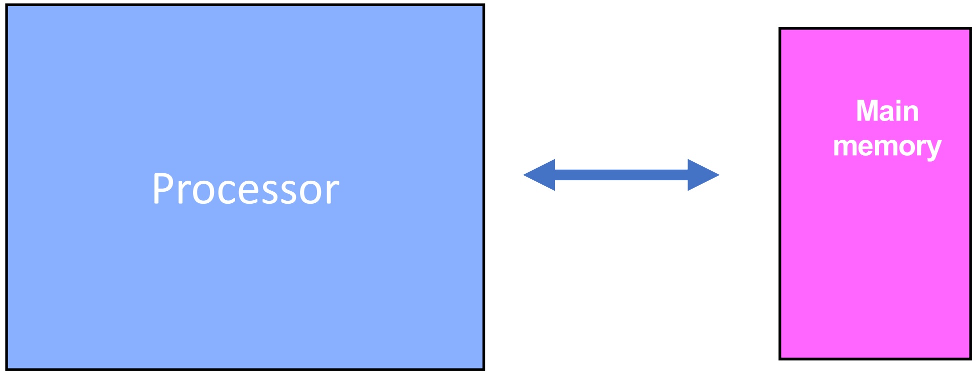
Why spend so long on memory?

- Cost of memory access often overlooked
- Simple cost model: memory accesses (reading/writing to RAM) has two costs:
  - **Latency:** cost to load/store a **single** “word” of memory
  - **Bandwidth:** average rate (bytes/sec) to load/store a **large chunk** of memory. We will use inverse time/byte)



Why spend so long on memory?

- Cost of memory access often overlooked
- Simple cost model: memory accesses (reading/writing to RAM) has two costs:
  - **Latency:** cost to load/store a **single** "word" of memory (α)
  - **Bandwidth:** average rate (bytes/sec) to load/store a **large chunk** of memory. We will use β : inverse time/byte)
- Cost = $\alpha + \beta * n$, where n is the number of "words" of memory transferred.



Bandwidth vs latency

Bandwidth

≈ data throughput (bits/second)

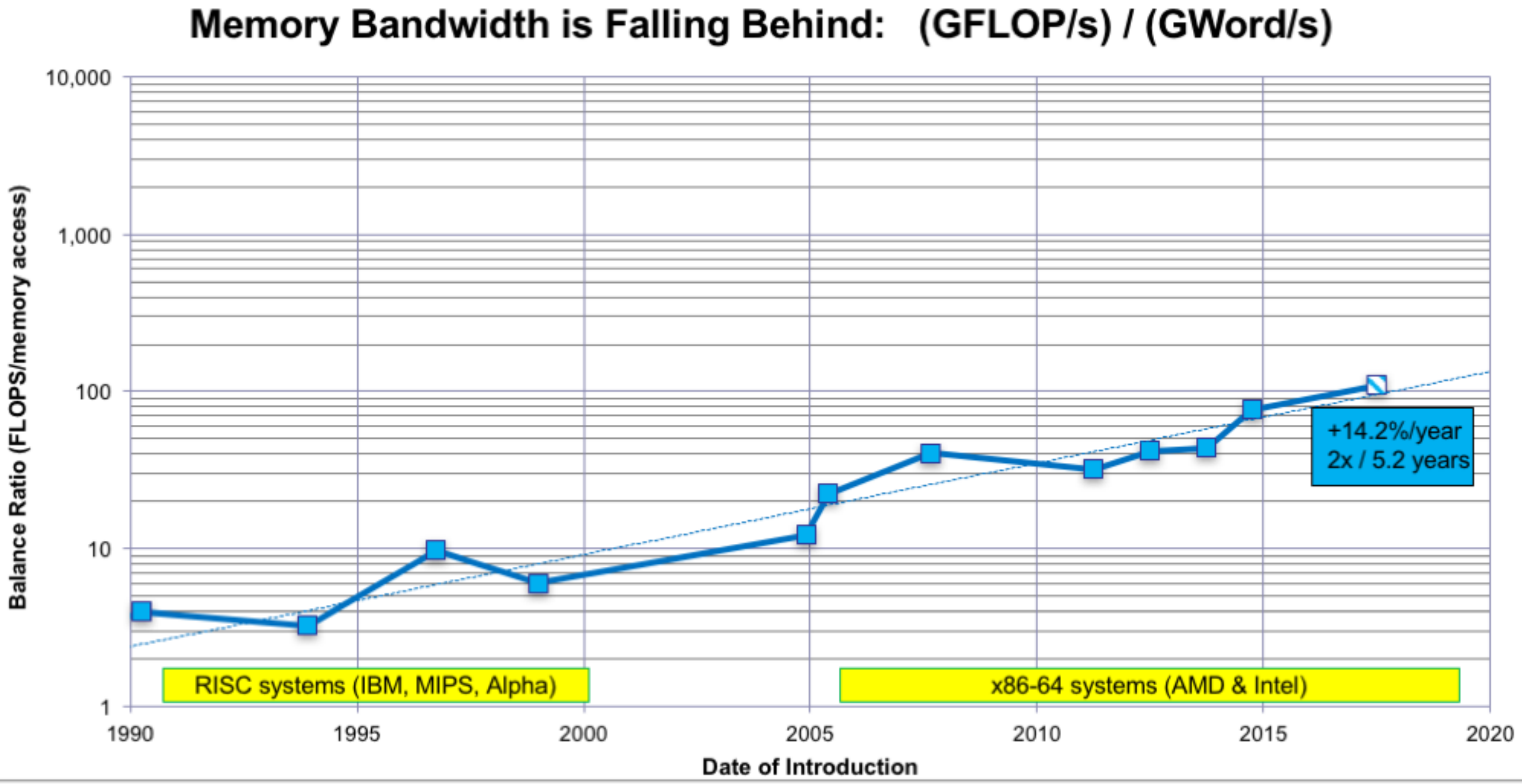
Latency

≈ delay due data travel time (ms)

Image: Katie Hempenius

Memory is slow relative to flops

Y-axis = cost of memory bandwidth and memory latency relative to arithmetic operations.



THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

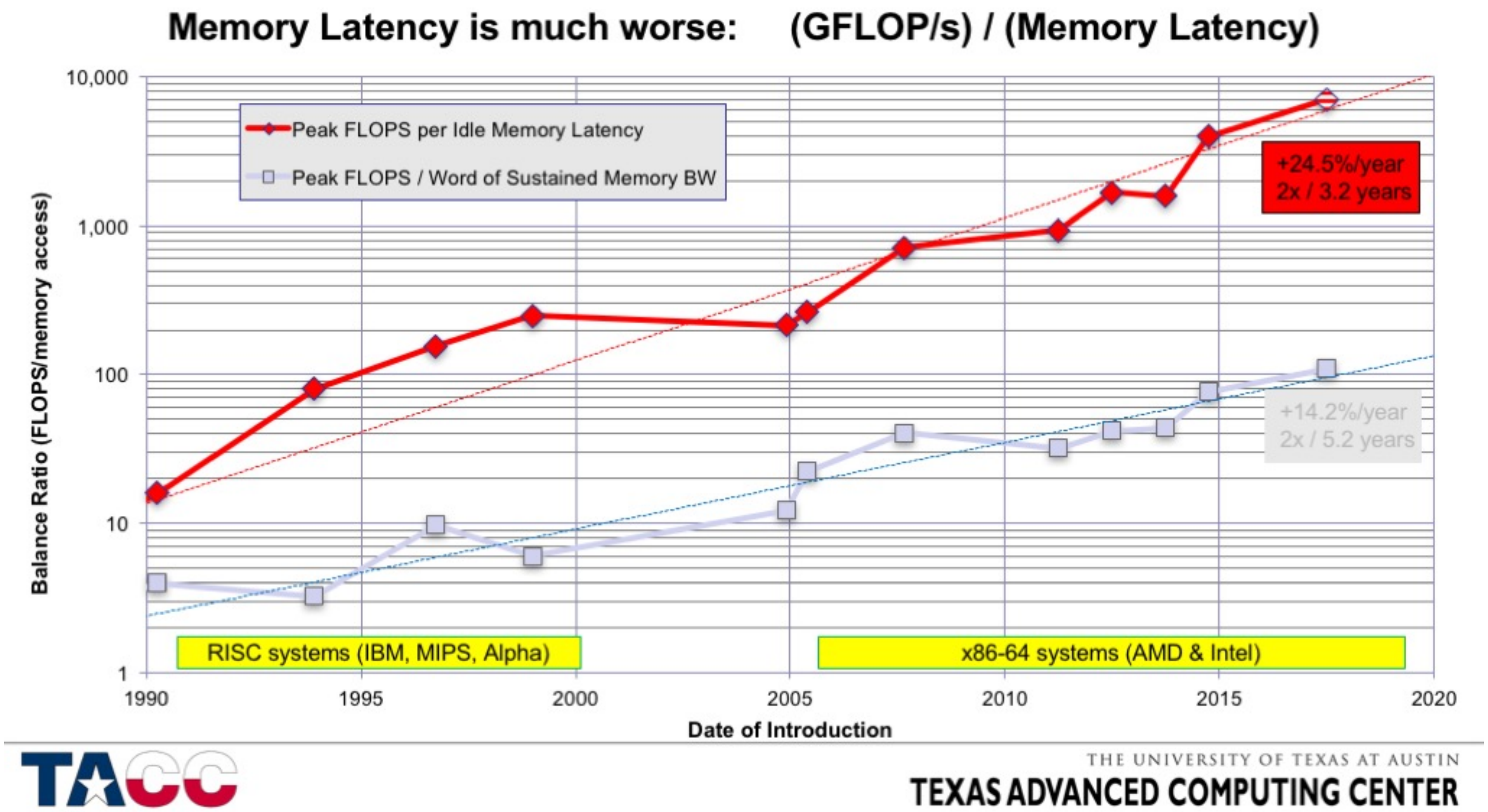
John McCalpin, SC16 invited talk

Memory is slow relative to flops

Y-axis = cost of memory bandwidth and memory latency relative to arithmetic operations.

Memory bandwidth and latency are both much slower than arithmetic operations!

HPC: “flops are free” is only true relative to memory costs.



John McCalpin, SC16 invited talk

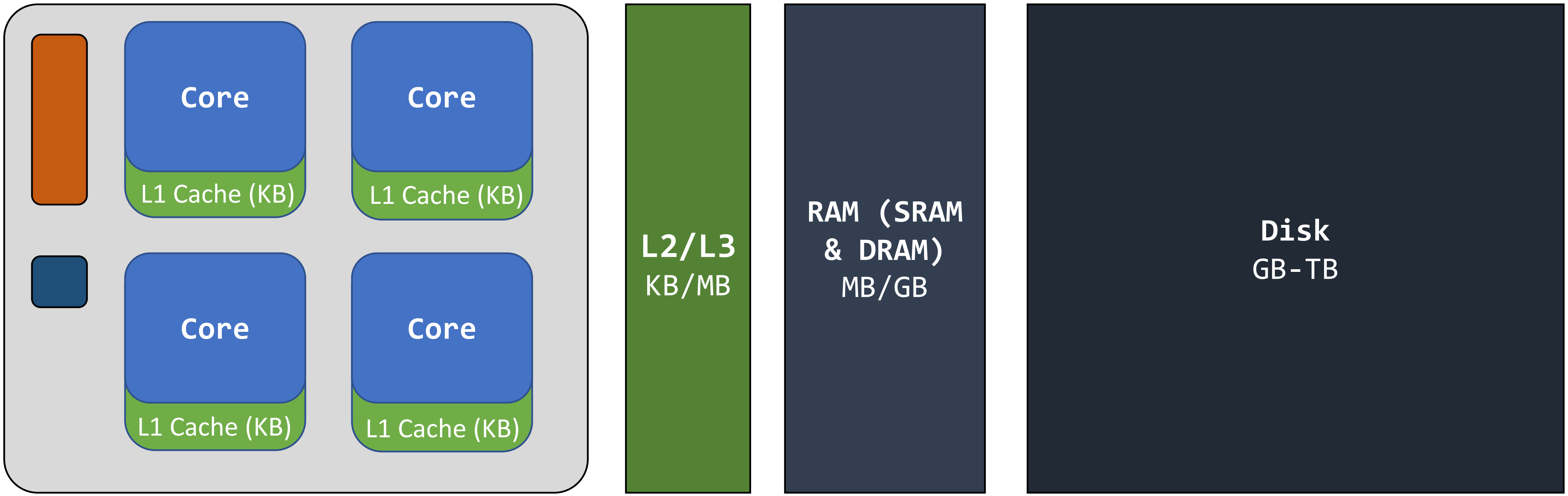
Smart Computer Architecture

Accessing memory can constitute the majority of a program's runtime

- **Naïve solution:** make the whole computer faster
  - Makes the whole computer more expensive too
  - Physical limitations, tight manufacturing tolerances
- **Smarter solution:** Use the notion of ***priority*** to help balance performance and cost
  - More commonly accessed memory should be accessed faster
  - Have multiple memory staging areas with differing performance

Computer Architecture: Caches

Idealized architecture



Smart Computer Architecture

Modern computers use a *memory hierarchy* to balance performance and cost

- Typically have register memory, L1, L2, maybe L3 caches.
- The closer to the cores (the chips that actually do work), the faster, but also smaller the caches become:
  - **Register memory:** on each core, specialized functionality, ~64 bytes
  - **L1 cache:** built into each core; couple hundred KB; ~100x faster than RAM
  - **L2 cache:** may be per CPU or per core; hundred of KB to MB; ~25x RAM
  - **L3 cache:** one per CPU; hundreds of KB to dozens of MB;
  - **RAM:** where program memory lives (stack and heap); several GBs
  - **Disk:** long term memory where files live; GBs, up to TBs

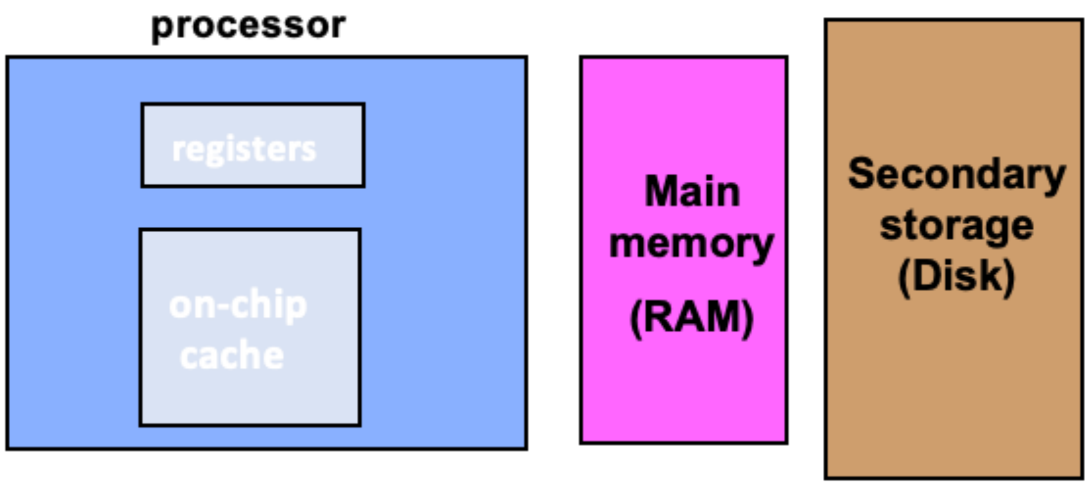
Data and temporal locality

Memory hierarchy is usually fast because programs usually exhibit **locality**

- If you touch a variable, it is likely that you will
  - use it again soon (**temporal locality**)
  - access its neighbors next (**spatial locality**)
- Data is not read from memory one word at a time. Instead, chunks (“cache lines”) are copied to L1, L2, etc caches.
  - How the caches are filled depends on the compiler/hardware
- Temporal and spatial locality makes it less likely that a cache line gets *evicted*.

Difference in speed

Hierarchy speeds up most common cases

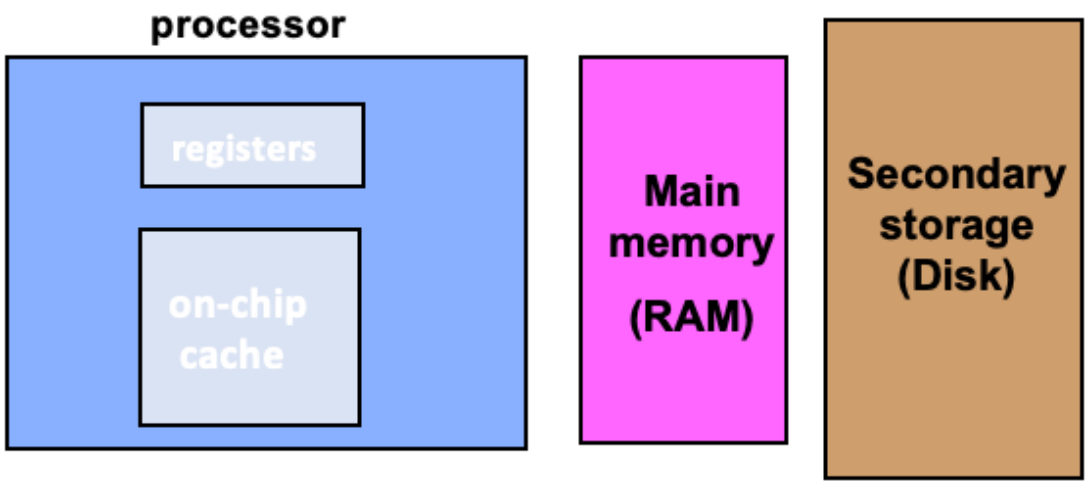


Speed	1ns	10-100ns	10ms
Size	KB	MB-GB	TB

Image adapted from Jim Demmel

Smart Computer Architecture

Hierarchy speeds up most common cases



Pick up a pencil on your desk

Grab a box of pencils from drawer

Buy a case of pencils at Costco

Caches Hits and Misses

Caching CAN drastically improve computer/program performance, but depends on the program.

- Is the variable we need next in cache or not?
  - If a variable is in cache, it's a *cache hit*
  - If a variable is not in cache, it's called a *cache miss*, and we have to read from a lower cache or RAM/disk
- Cache misses are expensive: reading from lower cache or RAM takes 10-100x longer than reading from L1

Memory hierarchy in practice

- Tools like *perf* in Linux can analyze cache misses. However, controlling what is placed into cache or register memory can get complicated.
- Often easier to use the memory hierarchy model as a conceptual tool.
- **The way we access memory** will change the performance of the program

Contiguous Memory and Stride

- Walking (i.e. incrementing) along one index may yield smaller steps in memory than other indices
- The number of elements/memory stepped over per increment in a given dimension is that dim's ***stride***
- The dimension with the smallest stride is the ***fastest dimension***
- ***Recall caching:*** caching grabs *contiguous* chunks of memory -> walking along the fastest dimension first will yield the most cache hits -> faster code

Bytes ->	1	2	3	4	5	6	7	8	9	10	11	12
x	x[0]				x[1]				x[2]			

Stride and Performance

- How you traverse an array can impact performance
- Traversal 1 will be faster than 2 **if n is large enough** to expose caching effects (assume A is row major)

Traversal 1:

```
int i, j;
for(i = 0; i < n; i++){
    for(j = 0; j < n; j++) {
        do_stuff(A[i][j]);
    }
}
```

Traversal 2:

```
int i, j;
for(j = 0; j < n; j++){
    for(i = 0; i < n; i++) {
        do_stuff(A[i][j]);
    }
}
```

mD Arrays: Direct Declaration

C++

```cpp
// Direct declaration of a 3D array
int A[2][4][8];
int A[][]…[];

// Can also do an initializer list with appropriate
// nesting of {}'s
```

mD Arrays: Dynamic Allocation

- There are three methods for allocating mD arrays dynamically
- Method 1 ensure contiguous memory
  - You won’t be able to use A[i1][i2]…[im] indexing though
- Method 2 ensures you can use traditional indexing
  - You probably won’t have contiguous memory though
- Method 3 blends 1 and 2 (cont. mem, traditional indexing)
  - It’s more effort than its worth

mD Arrays: Dynamic Allocation

- **Method 1:** Allocate an appropriately sized 1D array and handle the indexing yourself

```
int* A = NULL;
A = new int[n1*n2*…*nm];
i_flat = i1*(n2*…*nm) + i2*(n3*…*nm) + … + i(m-1)*nm + im;

// A[i_flat] = A[i1][i2]…[im];
```

mD Arrays: Dynamic Allocation

- **Method 2:** Use intermediate pointer arrays

```
// For a 2D array:
int** A = NULL;
A = new int*[n1]; // Allocates an array of n1 pointers
for(int i = 0; i < n1; i++) {
    A[i] = new int[n2]; // Note A[i] points to an array of size n2
}

// Can use A[i][j] now
```

mD Arrays: Dynamic Allocation

- **Method 3:** Allocate 1D to get contiguous memory, but also use intermediate pointer arrays

```
// For a 2D array:
int** A = NULL;
int* A_base = NULL;
A_base = new int[n1 * n2]; // Allocates an 1D array
A = new int*[n1]; // Allocates the row pointers
for(int i = 0; i < n1; i++) {
    A[i] = &A_base[i*n2]; // Note A[i] is a pointer
}
// Now A[i][j] = A_base[j + i*n2]
```

Pros and Cons of mD Dyn. Methods

- Contiguous memory helps with caching
  - Methods 1 & 3 yield contiguous memory
  - You also need the intermediate array for Method 3
- Method 2 also allows you to declare a ***ragged array,*** more flexibility.
- A lot of folks still use Method 1 (flat storage)