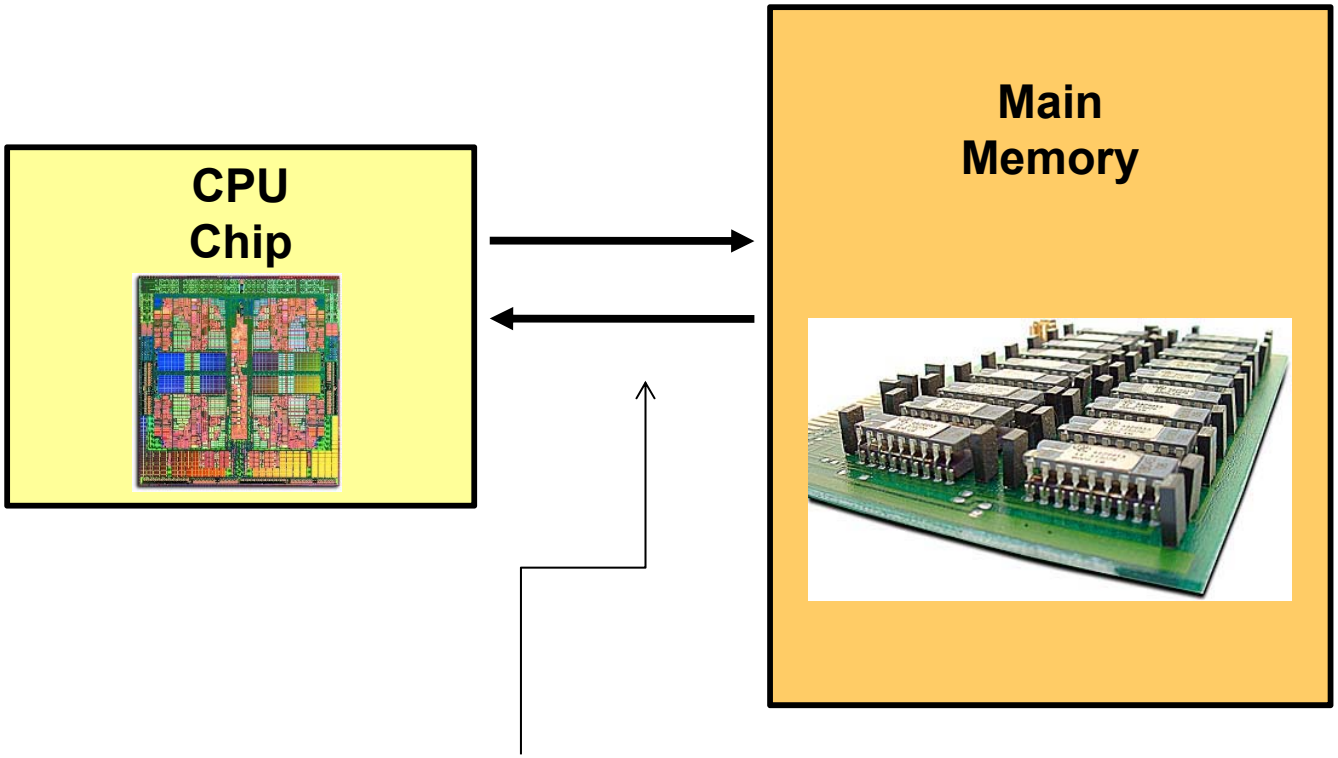
Problem: The Path Between a CPU Chip and Off-chip Memory is Slow

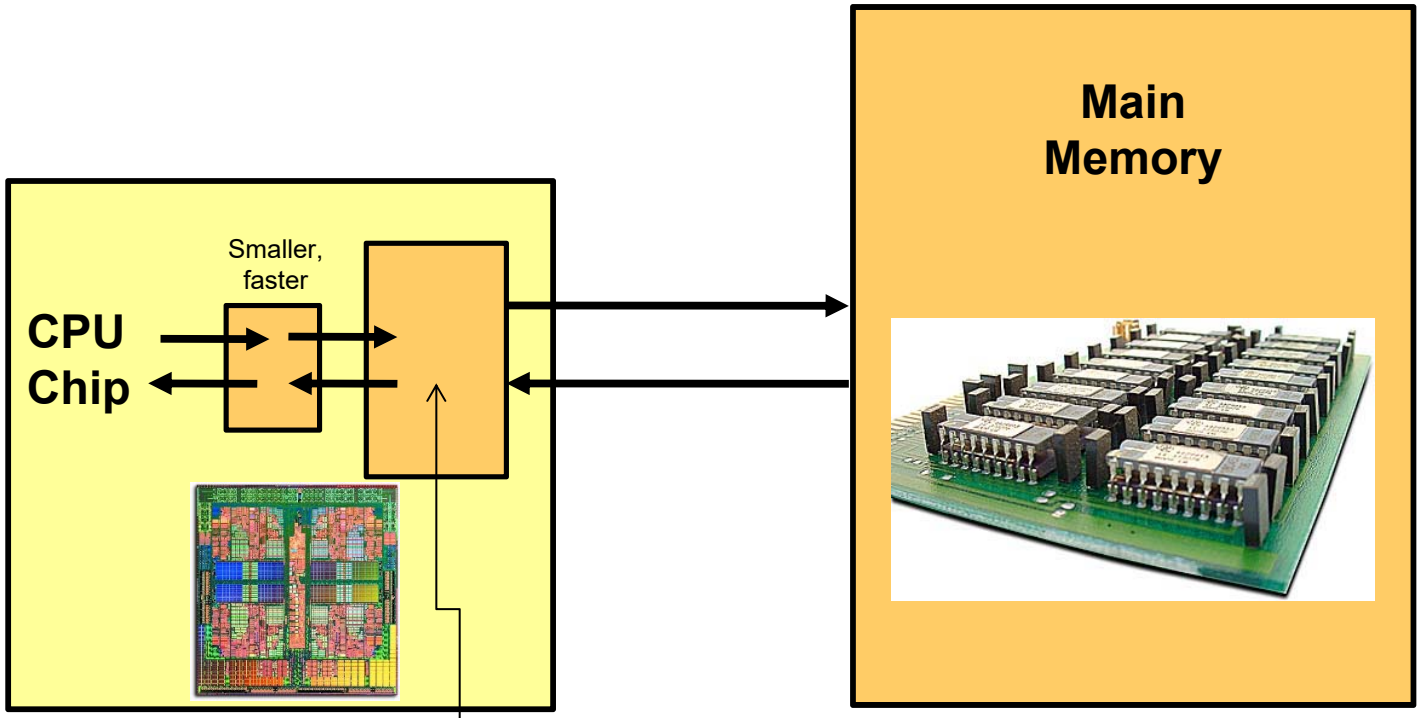


This path is relatively slow, forcing the CPU to wait for up to 200 clock cycles just to do a store to, or a load from, memory.

Depending on your CPU's ability to process instructions out-of-order, it might go idle during this time.

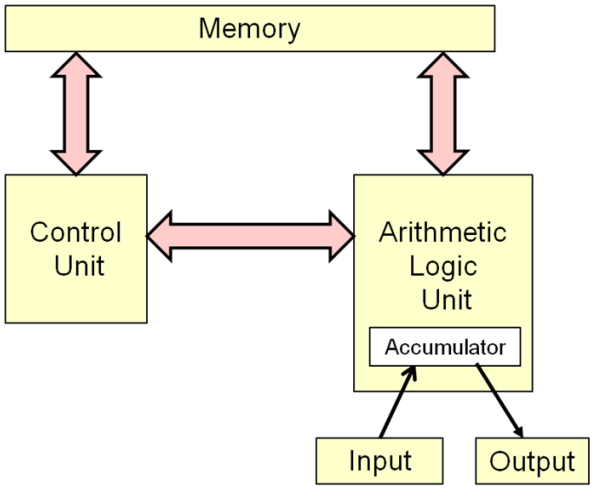
This is a *huge* performance hit!

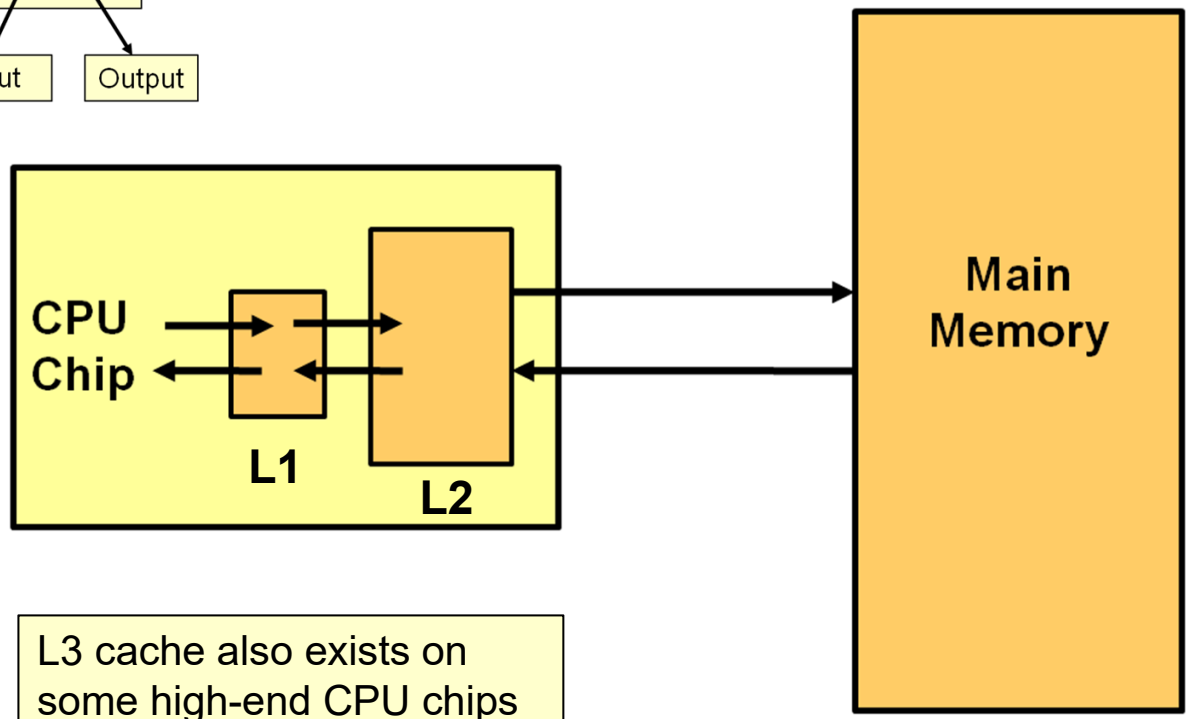
Solution: Hierarchical Memory Systems, or "Cache"



The solution is to add intermediate memory systems. The one closest to the CPU is small and fast. The memory systems get slower and larger as they get farther away from the CPU.

Cache and Memory are Named by "Distance Level" from the ALU





L3 cache also exists on some high-end CPU chips

Storage Level Characteristics

	L1	L2	Memory	Disk
Type of Storage	On-chip	On-chip	Off-chip	Disk
Typical Size	< 100 KB	< 8 MB	< 10 GB	Many GBs
Typical Access Time (ns)	.25 - .50	.5 – 25.0	50 - 250	5,000,000
Scaled Access Time	1 second	33 seconds	7 minutes	154 days
Bandwidth (MB/sec)	50,000 – 500,000	5,000 – 20,000	2,500 – 10,000	50 - 500
Managed by	Hardware	Hardware	OS	OS

Adapted from: John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, 2007. (4th Edition)

Usually there are two L1 caches – one for Instructions and one for Data. You will often see this referred to in data sheets as: “L1 cache: 32KB + 32KB” or “I and D cache”

Cache Hits and Misses

When the CPU asks for a value from memory, and that value is already in the cache, it can get it quickly.
This is called a ***cache hit***

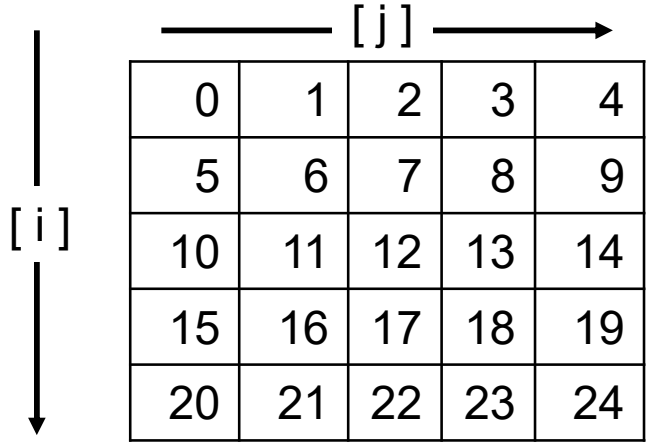
When the CPU asks for a value from memory, and that value is not already in the cache, it will have to go off the chip to get it.
This is called a ***cache miss***

While cache might be multiple kilo- or megabytes, the bytes are transferred in much smaller quantities, each called a **cache line.** The size of a cache line is typically just **64 bytes**.

Performance programming should strive to avoid as many cache misses as possible. That's why it is very helpful to know the cache structure of your CPU.

How Bad Is It? -- Demonstrating the Cache-Miss Problem

C and C++ store 2D arrays a row-at-a-time, like this, A[i][j]:



For large arrays, would it be better to add the elements by row, or by column? Which will avoid the most cache misses?

```
sum = 0.;
for( int i = 0; i < NUM; i++ )
{
        for( int j = 0; j < NUM; j++ )
        {
                float  f = ???
                sum += f;
        }
}
```

Sequential memory order → `float f = Array[ i ][ j ] ;`

Jump-around-in-memory order → `float f = Array[ j][ i ] ;`

Demonstrating the Cache-Miss Problem – Across Rows

```
#include <stdio.h>
#include <ctime>
#include <cstdlib>

#define NUM  10000

float   Array[NUM][NUM];

double  MyTimer( );

int
main( int argc, char *argv[ ] )
{
        float sum = 0.;
        double start  = MyTimer( );
        for( int i = 0; i < NUM; i++ )
        {
                for( int j = 0; j < NUM; j++ )
                {
                        sum += Array[ i ][ j ];              // access across a row
                }
        }
        double finish = MyTimer( );

        double row_secs = finish – start;
```

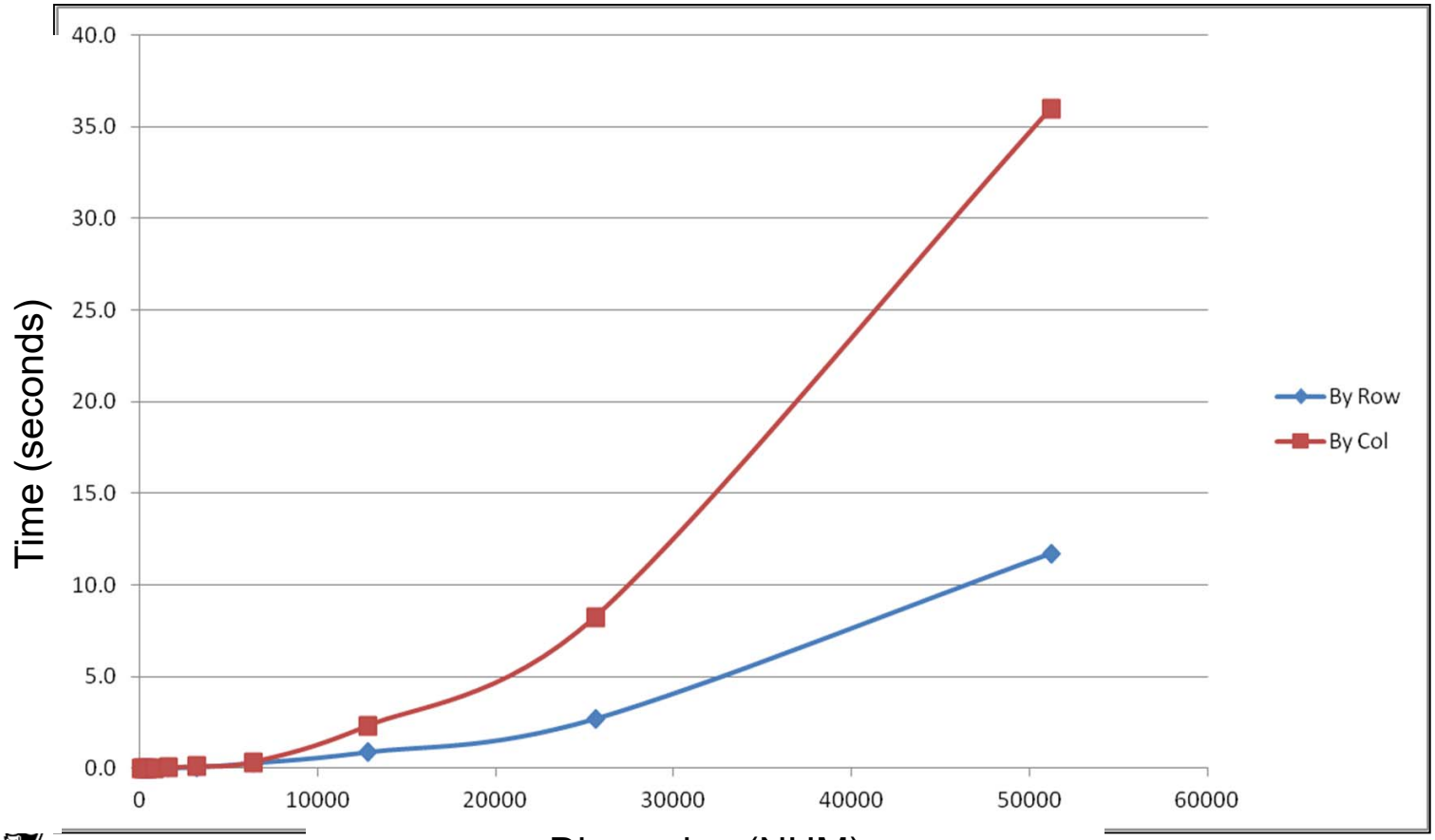
Demonstrating the Cache-Miss Problem – Down Columns

```
	sum = 0.;
	start  = MyTimer( );
	for( int i = 0; i < NUM; i++ )
	{
		for( int j = 0; j < NUM; j++ )
		{
			sum += Array[ j ][ i ];		// access down a column
		}
	}
	finish = MyTimer( );

	double col_secs = finish - start;
	fprintf( stderr, "NUM = %5d ; By rows = %lf ; By cols = %lf\n",
					NUM, row_secs, col_secs );
}
```

Oregon State University
Computer Graphics

Demonstrating the Cache-Miss Problem

Time, in seconds, to compute the array sums, based on by-row versus by-column order:



Array-of-Structures vs. Structure-of-Arrays:

```
struct xyz
{
        float x, y, z;
} Array[N];
```

X0 Y0 Z0
X1 Y1 Z1
X2 Y2 Z2
X3 Y3 Z3

```
float X[N], Y[N], Z[N];
```

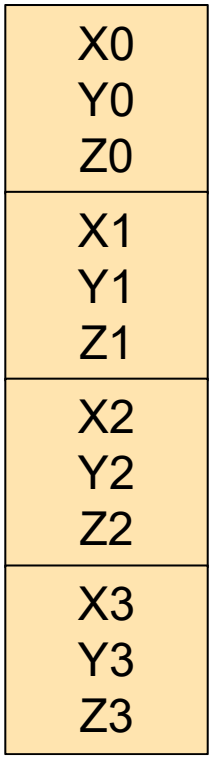
X0 X1 X2 X3 . . .
Y0 Y1 Y2 Y3 . . .
Z0 Z1 Z2 Z3 . . .

1. Which is a better use of the cache if we are going to be using X-Y-Z triples a lot?

2. Which is a better use of the cache if we are going to be looking at all X's, then all Y's, then all Z's?

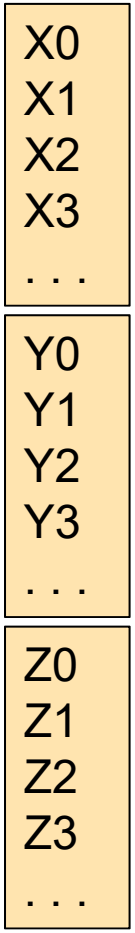
I've seen some programs use a "Shadow Data Structure" to get the advantages of both AOS and SOA

Computer Graphics is often a Good Use for Array-of-Structures:




```
struct xyz
{
        float x, y, z;
}  Array[N];

. . .

glBegin( GL_LINE_STRIP );
for( int i = 0; i < N; i++ )
{
        glVertex3f( Array[ i ].x, Array[ i ].y, Array[ i ].z );
}
glEnd( );
```

A Good Use for Structure-of-Arrays:

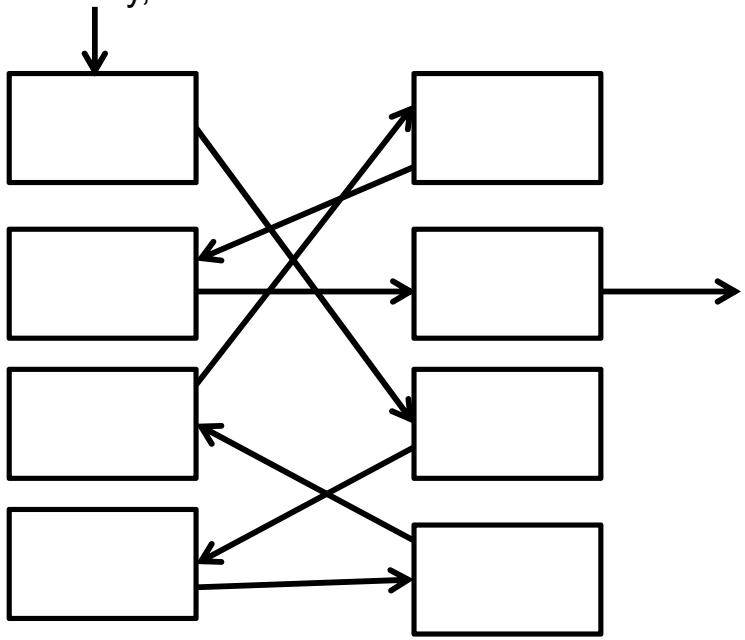



```
float X[N], Y[N], Z[N];
float Dx[N], Dy[N], Dz[N];
. . .

Dx[0:N] = X[0:N]  -  Xnow;
Dy[0:N] = Y[0:N]  -  Ynow;
Dz[0:N] = Z[0:N]  -  Znow;
```

Good Object-Oriented Programming Style can sometimes be Inconsistent with Good Cache Use:

```
class xyz
{
    public:
            float x, y, z;
            xyz *next;
            xyz( );
            static xyz *Head = NULL;
};

xyz::xyz( )
{
            xyz * n = new xyz;
            n->next = Head;
            Head = n;
};
```

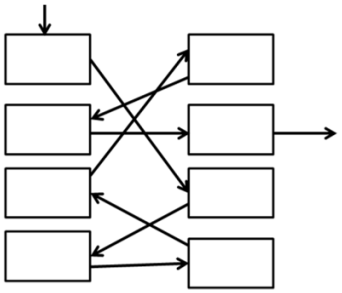
This is good OO style – it encapsulates and isolates the data for this class. Once you have created a linked list whose elements are all over memory, is it the best use of the cache?

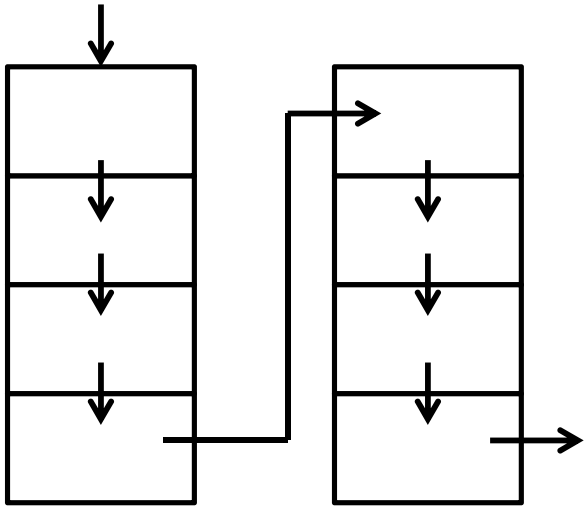


But, Here Is a Compromise:

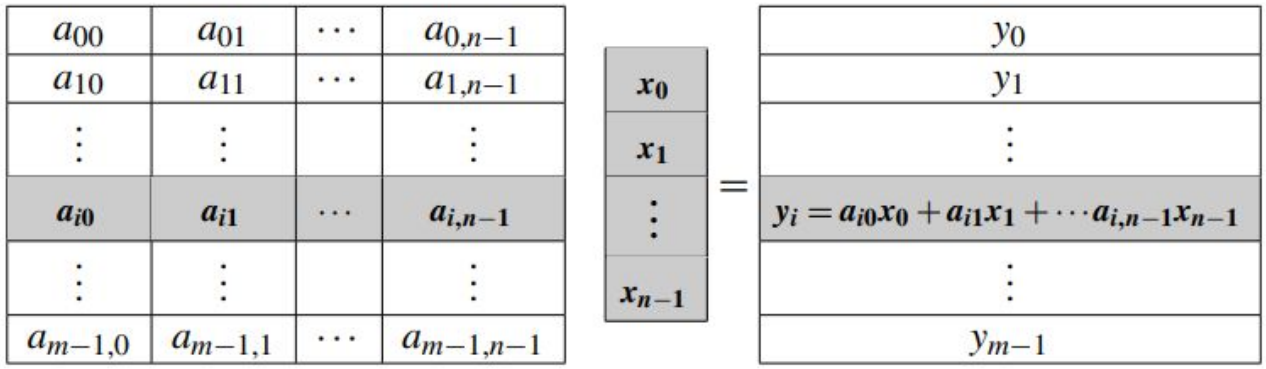
It might be better to create a large array of xyz structures and then have the constructor method pull new ones from that list. That would keep many of the elements close together while preserving the flexibility of the linked list.

When you need more, allocate another large array and link to it.





Matrix vector multiplication




```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

Performance numbers

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Observation 1

Threads	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Explanation 1

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- A **write-miss** occurs when a core tries to update a variable that's not in cache, and it has to access the main memory
- 8,000,000 x 8 shows more cache write-misses than either of the other inputs
- Bulk of these occur in Line 4
- Since the number of elements in the vector y is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, so line 4 slows down the execution of the program with the 8,000,000 × 8 input

Observation 2

Threads	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Explanation 2

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- A **read-miss** occurs when a core tries to read a variable that's not in cache, and it has to access main memory
- 8 x 8,000,000 shows more cache read-misses than either of the other inputs
- Bulk of these occur in Line 6
- for this matrix dimension, x has 8,000,000 elements, versus only 8000 or 8 for the other inputs