

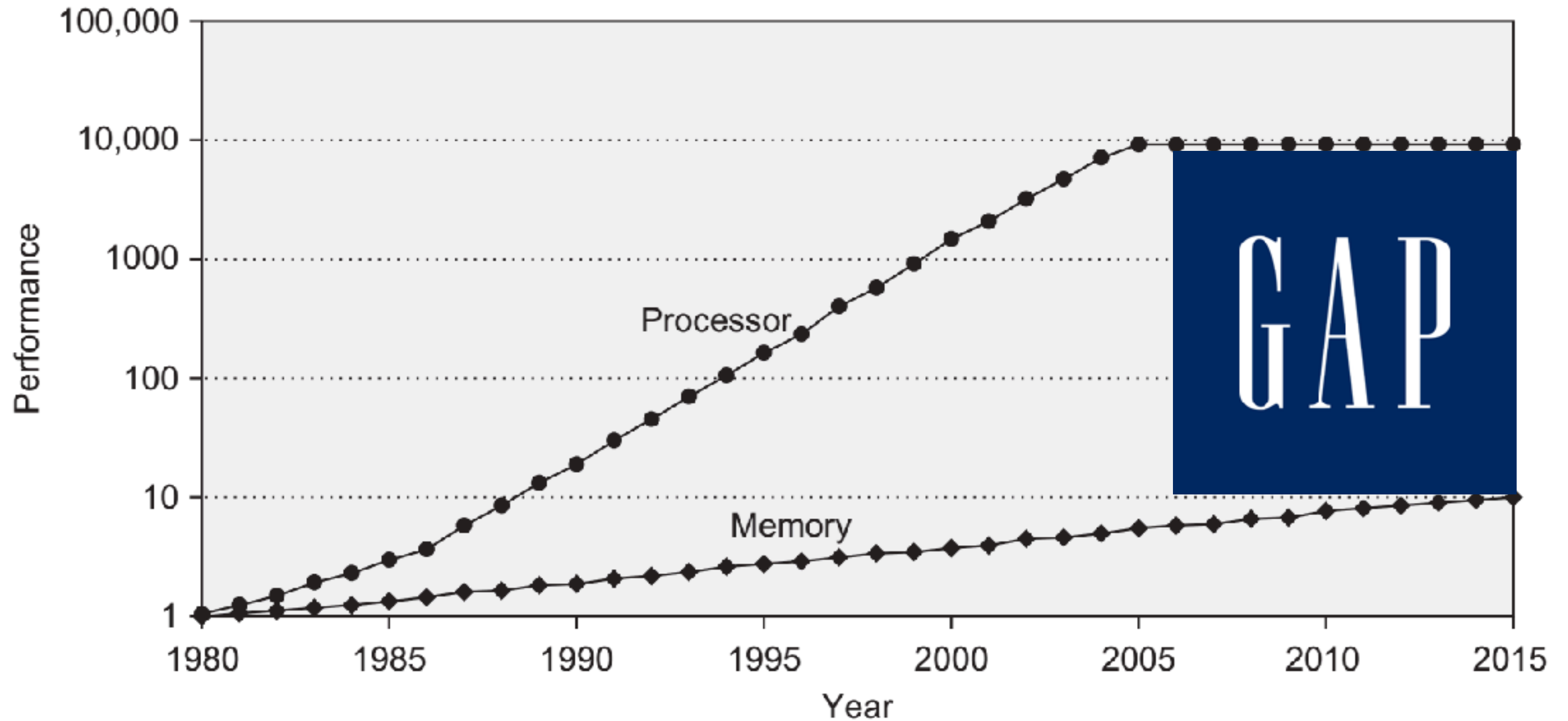
Memory Hierarchy Inside Out:

(5) Cache misses and their remedies

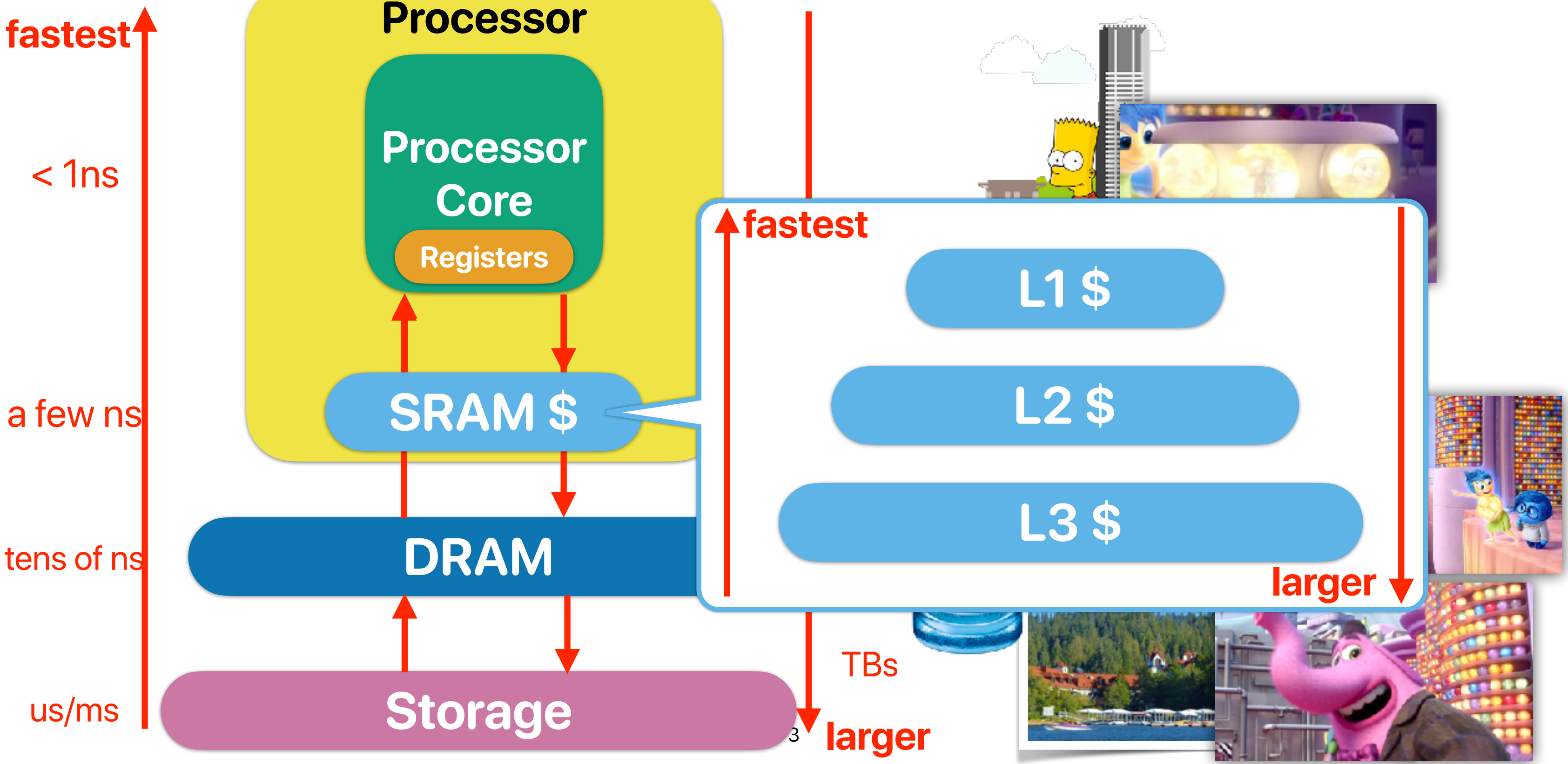
(III)

Hung-Wei Tseng

Recap: Performance gap between Processor/Memory



Recap: Memory Hierarchy



Review: C = ABS

- **C**: Capacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
 - N-way: N blocks in a set, A = N
 - 1 for direct-mapped cache
- **B**: Block Size (Cacheline)
 - How many bytes in a block
- **S**: Number of **S**ets:
 - A set contains blocks sharing the same index
 - 1 for fully associate cache
- number of bits in **b**lock offset — $\lg(\mathbf{B})$
- number of bits in **s**et index: $\lg(\mathbf{S})$
- tag bits: $\text{address_length} - \lg(\mathbf{S}) - \lg(\mathbf{B})$
 - address_length is 64 bits for 64-bit machine

$$\frac{\text{address}}{\text{block_size}} \pmod{S} = \text{set index}$$

memory address:



Review: 3Cs of misses

- Compulsory miss
 - Cold start miss. First-time access to a block
- Capacity miss
 - The working set size of an application is bigger than cache size
- Conflict miss
 - Required data replaced by block(s) mapping to the same set
 - Similar collision in hash

Recap: Summary of Optimizations

- Hardware
 - Prefetch — compulsory miss
 - Write buffer — miss penalty
 - Bank/pipeline — miss penalty
 - Critical word first and early restart — miss penalty
- Software
 - Data layout to improve locality — compulsory/conflict/capacity misses

Array of structures or structure of arrays

	Array of objects	object of arrays									
	<pre>struct grades { int id; double *homework; double average; };</pre>	<pre>struct grades { int *id; double **homework; double *average; };</pre>									
ID *homework average ID *homework average		<table><tr><td>ID</td><td>ID</td><td>ID</td></tr><tr><td>homework</td><td>homework</td><td>homework</td></tr><tr><td>average</td><td>average</td><td>average</td></tr></table>	ID	ID	ID	homework	homework	homework	average	average	average
ID	ID	ID									
homework	homework	homework									
average	average	average									
average of each homework	<pre>for(i=0;i<homework_items; i++) { gradesheet[total_number_students].homework[i] = 0.0; for(j=0;j<total_number_students;j++) gradesheet[total_number_students].homework[i] +=gradesheet[j].homework[i]; gradesheet[total_number_students].homework[i] /= (double)total_number_students; }</pre>	<pre>for(i = 0;i < homework_items; i++) { gradesheet.homework[i][total_number_students] = 0.0; for(j = 0; j <total_number_students;j++) { gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j]; } gradesheet.homework[i][total_number_students] /= total_number_students; }</pre>									

Outline

- The remedies of cache misses — the software version
- The basic of virtual memory

**How can programmer improve
memory performance?**

Loop interchange/fission/fusion

Demo — programmer & performance

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

$O(n^2)$

Complexity

$O(n^2)$

Same

Instruction Count?

Same

Same

Clock Rate

Same

Better

CPI

Worse

Loop interchange

NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
 - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
 - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];  
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */  
for(i = 0; i < 512; i++)  
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e  
for(i = 0; i < 512; i++)  
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

- A. ~10%
- B. ~20%
- C. ~40%
- D. ~80%
- E. 100%



What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
 - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];  
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */  
for(i = 0; i < 512; i++)  
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e  
for(i = 0; i < 512; i++)  
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

A. ~10%

B. ~20%

C. ~40%

D. ~80%

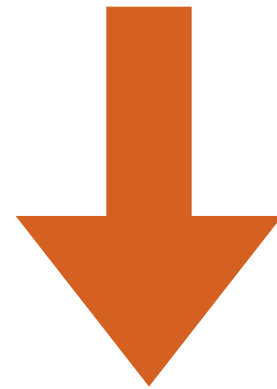
E. 100%

Loop fission

Loop fission

B

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```



Loop fission

A

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

What if we change the processor?

- If we have an intel processor with a 32KB, 8-way, 64B-blocked L1 cache, which version of code performs better?
 - A. Version A, because the code incurs fewer cache misses
 - B. Version B, because the code incurs fewer cache misses
 - C. Version A, because the code incurs fewer memory references
 - D. Version B, because the code incurs fewer memory references
 - E. They are about the same

A

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```


What if we change the processor?

- If we have an intel processor with a 32KB, 8-way, 64B-blocked L1 cache, which version of code performs better?
 - A. Version A, because the code incurs fewer cache misses
 - B. Version B, because the code incurs fewer cache misses
 - C. Version A, because the code incurs fewer memory references
 - D. Version B, because the code incurs fewer memory references**
 - E. They are about the same

A

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Loop optimizations

B

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Loop fission



A

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

A

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

Loop fusion



B

```
double a[8192], b[8192], c[8192], \
      d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Blocking/Tiling

Case study: Matrix Multiplication

```
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        for(k = 0; k < ARRAY_SIZE; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```

Algorithm class tells you it's $O(n^3)$

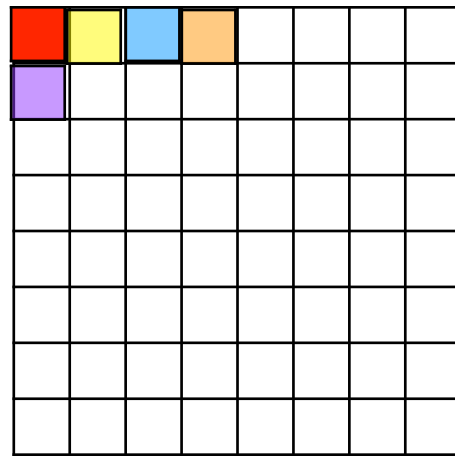
If $n=1024$, it takes about 1 sec

How long is it take when $n=2048$?

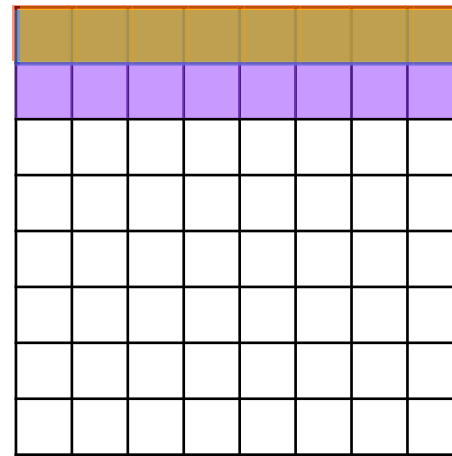
Matrix Multiplication

```
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        for(k = 0; k < ARRAY_SIZE; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```

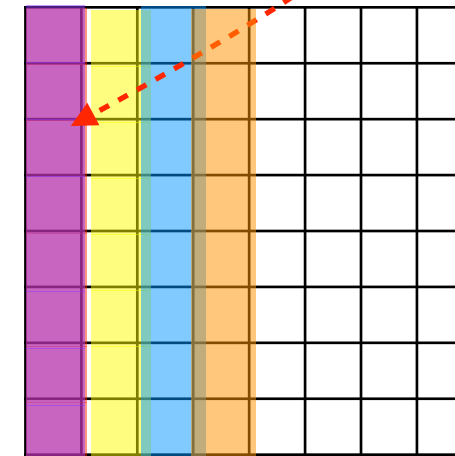
Very likely a miss if
array is large



c



a



b

- If each dimension of your matrix is 2048
 - Each row takes 2048×8 bytes = 16KB
 - The L1 \$ of intel Core i7 or AMD RyZen is 32KB, 8-way, 64-byte blocked
 - You can only hold at most 2 rows/columns of each matrix!
 - You need the same row when j increase!

It's show time!

Simply the addresses of B[]

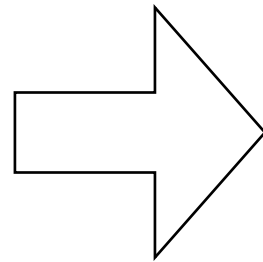
element	address	tag	index
b[0][0]	7F68297E1000	7F68297E1	0
b[1][0]	7F68297E1800	7F68297E1	20
b[2][0]	7F68297E2000	7F68297E2	0
b[3][0]	7F68297E2800	7F68297E2	20
b[4][0]	7F68297E3000	7F68297E3	0
b[5][0]	7F68297E3800	7F68297E3	20
b[6][0]	7F68297E4000	7F68297E4	0
b[7][0]	7F68297E4800	7F68297E4	20
b[8][0]	7F68297E5000	7F68297E5	0
b[9][0]	7F68297E5800	7F68297E5	20
b[10][0]	7F68297E6000	7F68297E6	0
b[11][0]	7F68297E6800	7F68297E6	20
b[12][0]	7F68297E7000	7F68297E7	0
b[13][0]	7F68297E7800	7F68297E7	20
b[14][0]	7F68297E8000	7F68297E8	0
b[15][0]	7F68297E8800	7F68297E8	20
b[16][0]	7F68297E9000	7F68297E9	0
b[17][0]	7F68297E9800	7F68297E9	20
b[18][0]	7F68297EA000	7F68297EA	0

We only used $17 \times 64 = 1088$ bytes!

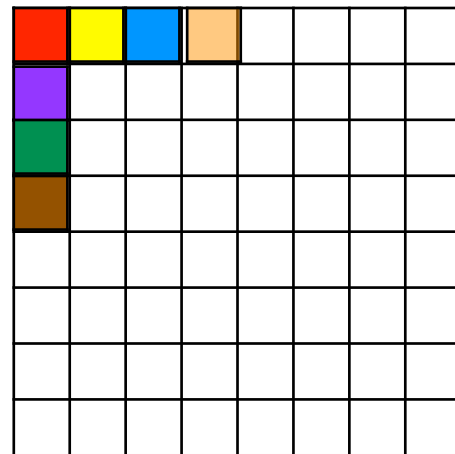
Conflict miss starts! 

Block algorithm for matrix multiplication

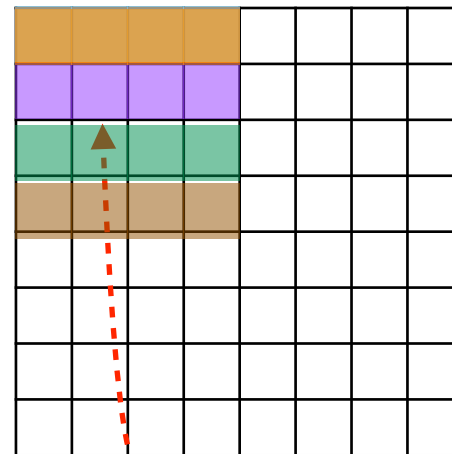
```
for(i = 0; i < ARRAY_SIZE; i++) {  
  for(j = 0; j < ARRAY_SIZE; j++) {  
    for(k = 0; k < ARRAY_SIZE; k++) {  
      c[i][j] += a[i][k]*b[k][j];  
    }  
  }  
}
```



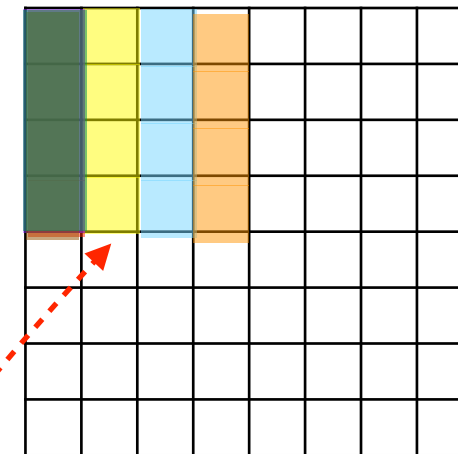
```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
            c[ii][jj] += a[ii][kk]*b[kk][jj];  
    }  
  }  
}
```



c



a



b

**You only need to hold these
sub-matrices in your cache**

What kind(s) of misses can block algorithm remove?

- Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an intel Core i7)

Naive

```
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        for(k = 0; k < ARRAY_SIZE; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```

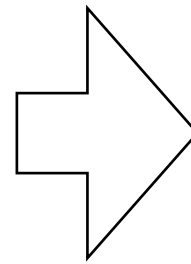
Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss**
- E. Compulsory & conflict miss

Matrix Transpose

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}
```

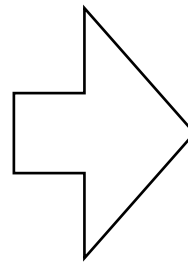


```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        b_t[i][j] += b[j][i];
    }
}

for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        // Compute on b_t
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
        }
    }
}
```

Matrix Transpose

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```



```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
        }  
    }  
}
```

What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

```
Block
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

Block + Transpose

```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        b_t[i][j] += b[j][i];
    }
}

for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        // Compute on b_t
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
        }
    }
}
```

What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss**
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

Block + Transpose

```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        b_t[i][j] += b[j][i];
    }
}

for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        // Compute on b_t
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
        }
    }
}
```

Tips of software optimizations

- Carefully layout your data structure can improve capacity misses!
- Make your data structures align with the access pattern can better exploit cache locality — improve conflict misses
- Implementing algorithms in a more cache friendly way!

Summary of Optimizations

- Software
 - Data layout — capacity miss, conflict miss, compulsory miss
 - Blocking — capacity miss, conflict miss
 - Loop fission — conflict miss — when \$ has limited way associativity
 - Loop fusion — capacity miss — when \$ has enough way associativity
 - Loop interchange — conflict/capacity miss
- Hardware
 - Prefetch — compulsory miss
 - Write buffer — miss penalty
 - Bank/pipeline — miss penalty
 - Critical word first and early restart — miss penalty

Let's dig into this code

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```


What will happen?

- If we execute the code on the right-hand side code on a machine with only 32 GB of physical memory installed and the dim is "70000" (requires $70000 \times 70000 \times 8$ bytes ~ 37 GB memory at least), What will happen?
 - A. The program will crash in one of the malloc function call
 - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
 - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
 - D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

What will happen?

- If we execute the code on the right-hand side code on a machine with only 32 GB of physical memory installed and the dim is "70000" (requires $70000 \times 70000 \times 8$ bytes ~ 37 GB memory at least), What will happen?
 - A. The program will crash in one of the malloc function call
 - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
 - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
 - D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

What will happen?

- If we execute the code on the right-hand side code on a machine with only 32 GB of physical memory installed and the dim is "70000" (requires $70000 \times 70000 \times 8$ bytes ~ 37 GB memory at least), What will happen?
 - A. The program will crash in one of the malloc function call
 - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
 - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
 - D. The program will finish without any issue**

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

Announcement

- Assignment #2 due this Thursday
 - Please start early — some programs take very long time to finish
 - As mentioned in the beginning of each assignment — server busy cannot be a reason for late submission or extension
 - If you have questions that cannot be address online, please come to office hours
 - Today: Hung-Wei's — 2:30p—3:30p and Kuan-Chieh's 3:30p-4:30p
- Midterm next Tuesday
 - You can only take the exam in-person, closed-book, closed-note
 - Please bring your student ID and we will check
 - You may review/focus on the materials/topics covered in lectures
 - You SHOULD review your assignments
 - Cover topics including this Thursday

We're not psychic readers

- Please make sure you show your “complete” process of thinking
- Please write down the equation you cite
- Don't expect we will assume you know those basic things

Computer Science & Engineering

203

つづく

