Overview of Mini-Project 1

Outline

- **CPU Achievable Peak Performance**
- Matrix-Multiplication Discussion
- Overview of Mini-Project 1
- A Quick Revie
  - SIMD Parallelis
  - OpenMP Pragmas
  - CPU Memory Hierarchy
  - False Sharing
- Mini-Project 1.1 - Matrix Multiply
- Mini-Project 1.2 - K-Means

Peak Single-precision Performance

- Intel Sandy Bridge 3.5 Ghz, 4 cores * 2 sockets

[GFLOPS]	1 core	2 cores	4 cores	8 cores
None, scalar				56
SSE				224
AVX				

Most CPU applications achieve a fraction of this

In addition …

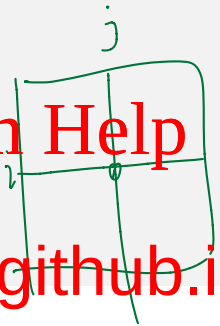
- Deep memory hierarchy
  - Registers
  - L1 cache
  - L2 cache
  - TLB (Translati
- Other forms of parallelism
  - Pipelining
  - Instruction-level parallelism (multiple functional units)
- "Free operations" are not free

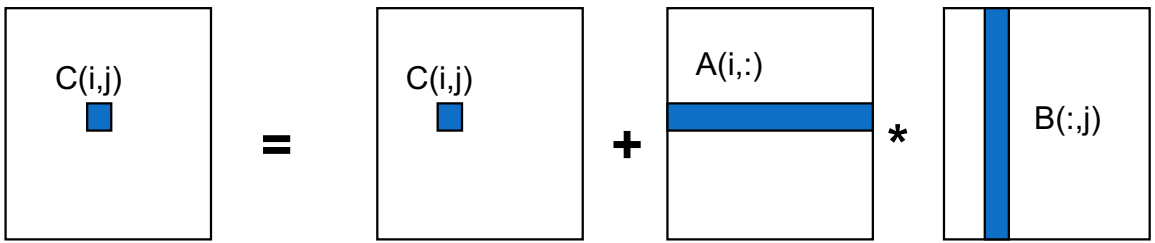
Outline

- CPU Achievable Peak Performance
- **Matrix-Multiplication Discussion**
- Overview of Mini-Project 1
- A Quick Revie
  - SIMD Parallelis
  - OpenMP Pragmas
  - CPU Memory Hierarchy
  - False Sharing
- Mini-Project 1.1 - Matrix Multiply
- Mini-Project 1.2 - K-Means

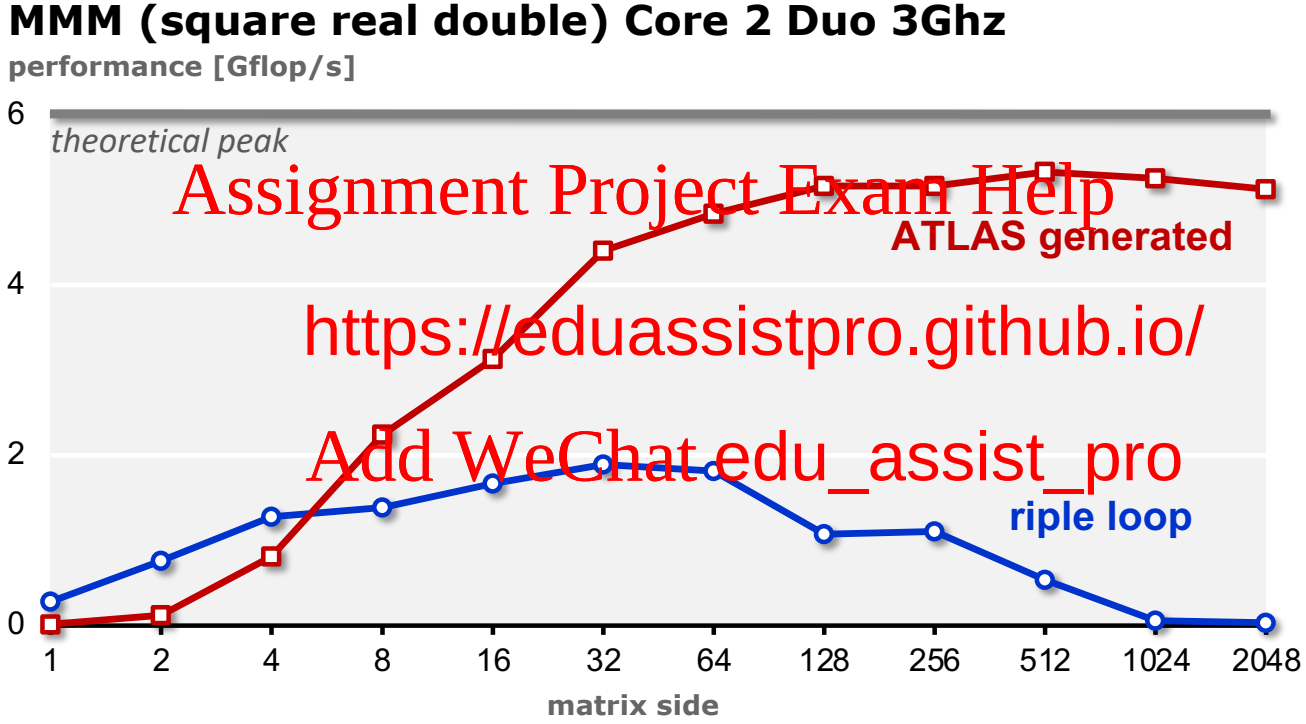
Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
```





Matrix-Matrix Multiplication Speed



- Huge performance difference for large sizes
- Great case study to learn memory hierarchy optimization

Note on Matrix Storage

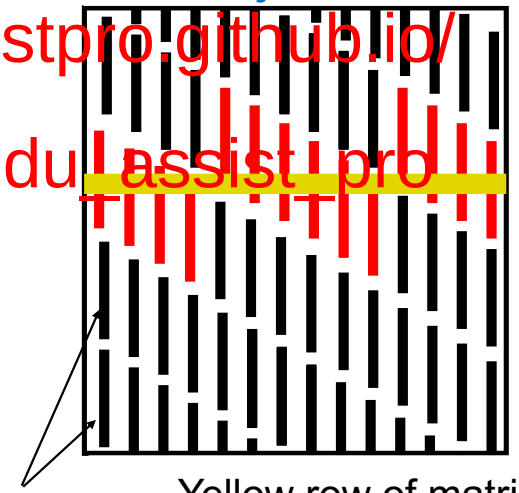
- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout
  - by column, or "column major" (Fortran default);
    `A[i,j]` at `i + j*n`
  - by row, or "row major" (C default)
    `A[i,j]` at `i*n`

Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

ajor matrix in memory



cachelines

Yellow row of matrix is stored in red cachelines

- Column major (for now)

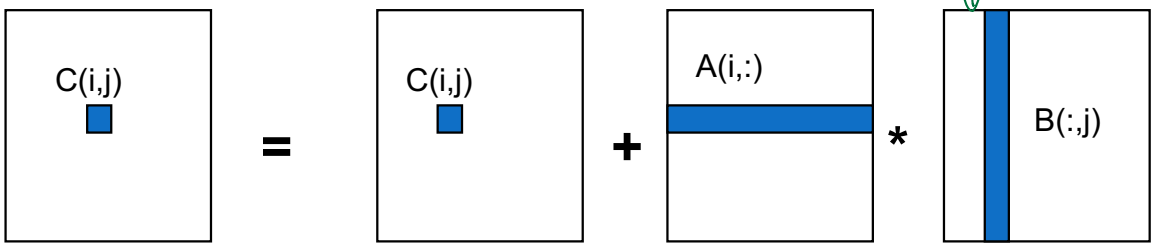
Naïve Matrix Multiply

row major.

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C(i,j) into f
      {read column j o
      for k = 1 to n
         C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {write C(i,j) back to slow memory}
```

A(i,:)

loads



Naïve Matrix Multiply

Number of slow memory references on unblocked matrix multiply

$m = n^3$ to read each column of B n times

$+ n^2$ to read each row of A once

$+ 2n^2$ to read and write each element of C once

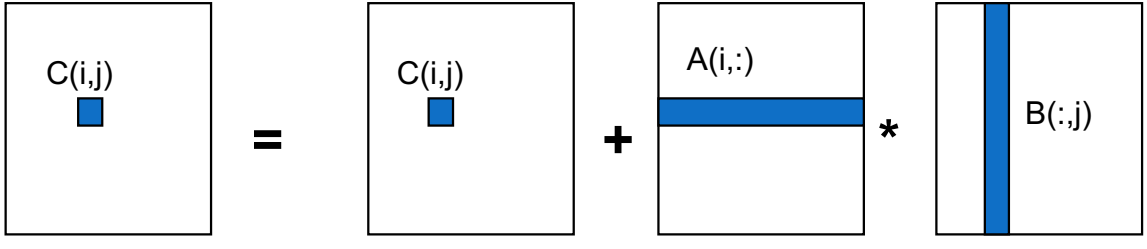
$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (n$

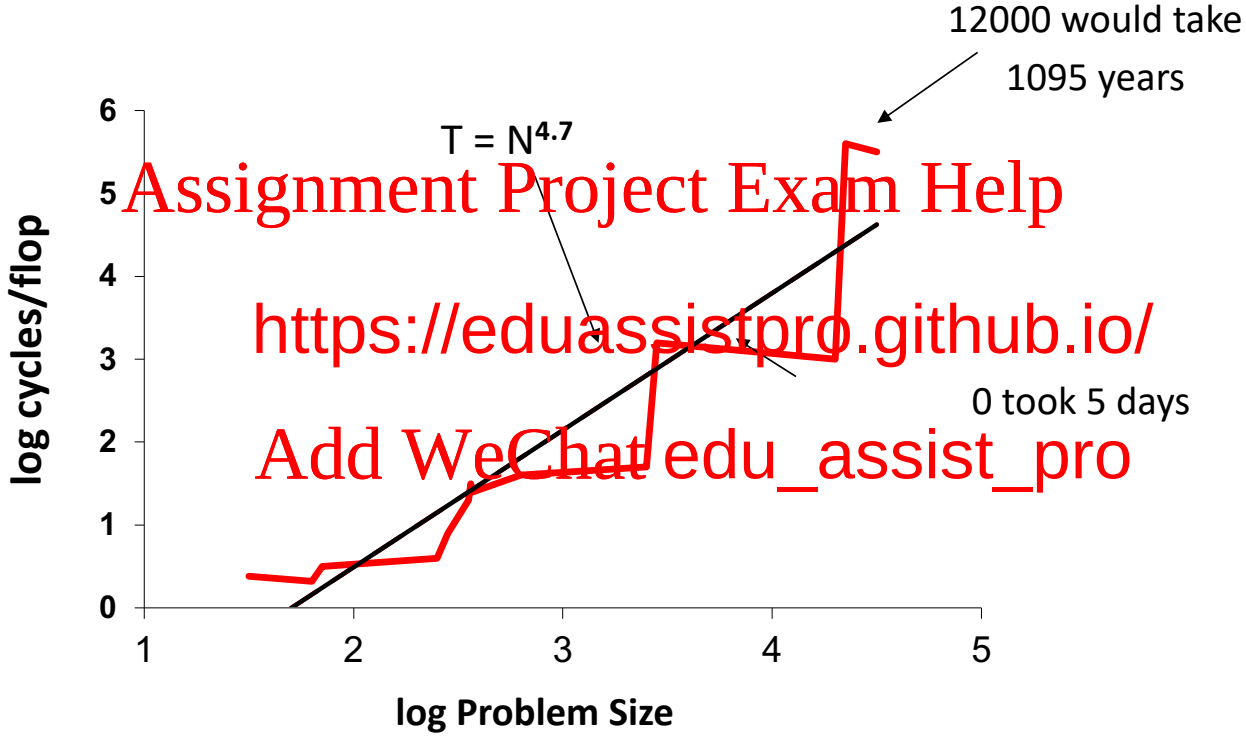
≈ 2 for large n, n itiply

Inner two loops are just matrix-vector multip mes B

Similar for any other order of 3 loops

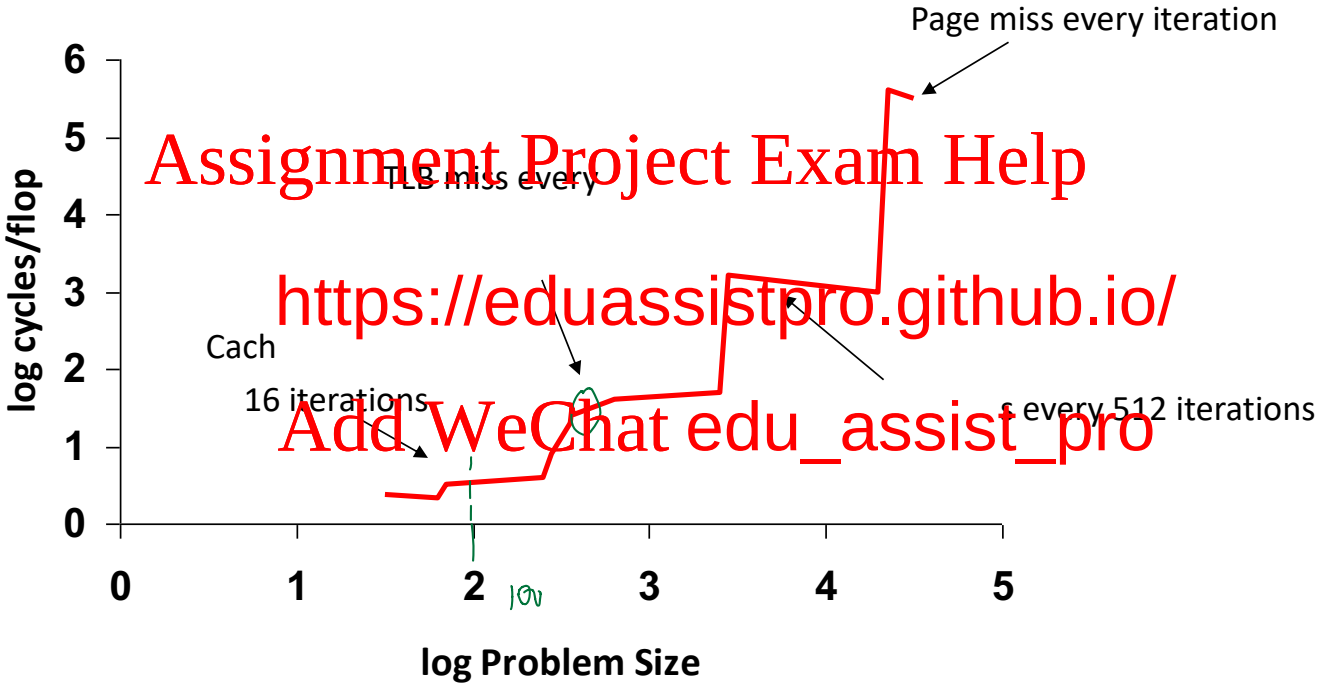


Naïve Matrix Multiply on RS/6000



O(N³) performance would have constant cycles/flop

Naïve Matrix Multiply on RS/6000



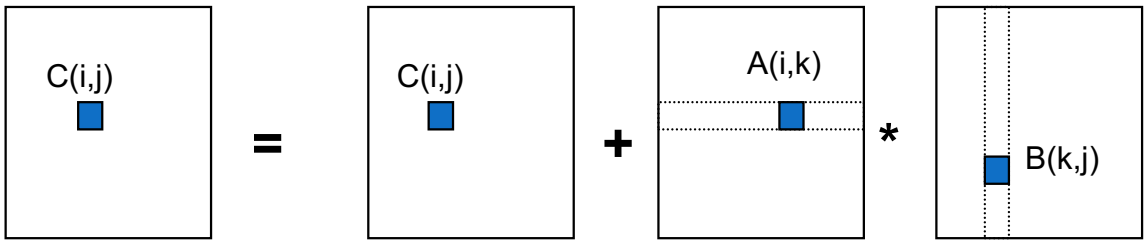
Slide source: Larry Carter, UCSD

Blocked (Tiled) Matrix Multiply

A,B,C = **NxN** matrices of **b x b** sub-blocks, **N = n / b**

```
for i = 1 to N
  for j = 1 to N
    {read block C(i,j) into fast memory}
    for k = 1 to
      {read b                                    (N^3) slow loads
      {read block B(k,j) into fast mem
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do b                n blocks}
    {write block C(i,j) back to slow me
```

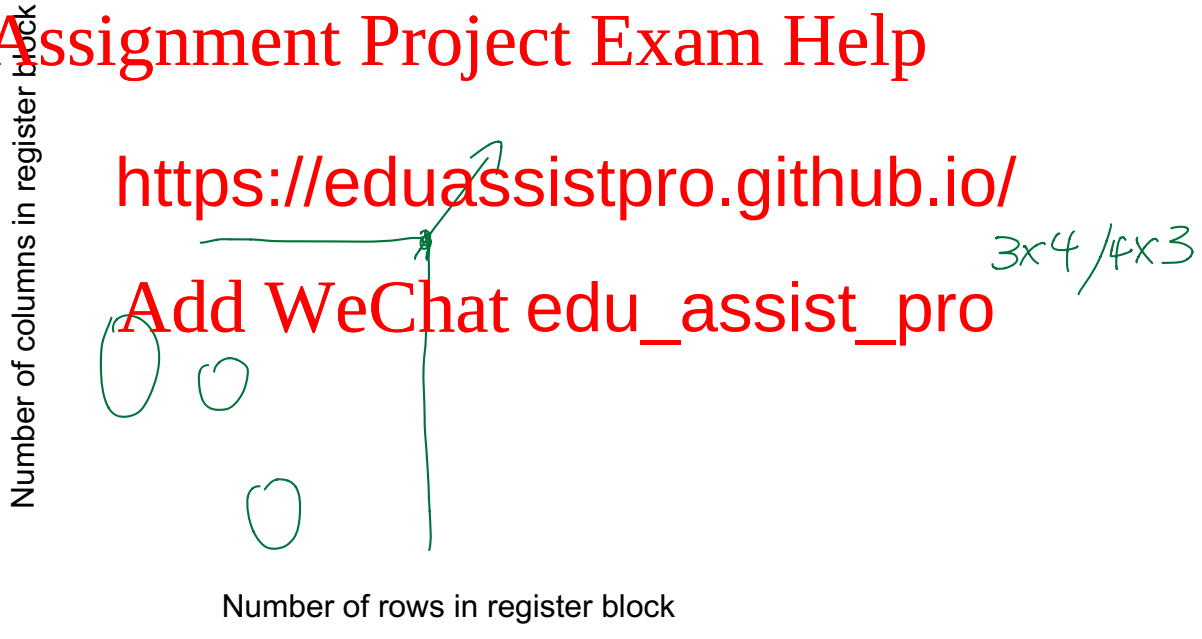
block should be smaller than cache & smallest



Further Matrix-Multiply Optimizations in Real-world Libraries

- Block size adaptation for appropriate caches
- **Register-level b**
- **Copy optimization (data layout)**
- **Optimizing the mini-matrix-multiply (**
- Multi-level blocking
- Multi-level copying

Register-level blocking



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.

Slide source: Jim Demmel, UCB

Copy optimization

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality
  - Alternative: u f users willing)

Original matrix
(numbers are addresses)

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

2

0	2	8	10
1	3	9	11
4	6	12	13
5	7	14	15

Slide source: Jim Demmel, UCB

Outline

- CPU Achievable Peak Performance
- Matrix-Multiplication Discussion
- **Overview of Mini-Project 1**
- A Quick Revie
  - SIMD Parallelis
  - OpenMP Pragmas
  - CPU Memory Hierarchy
  - False Sharing
- Mini-Project 1.1 - Matrix Multiply
- Mini-Project 1.2 - K-Means

Mini-Project 1

The goal of this project is to use your understanding of parallel computing resources in a multicore microprocessor to optimize two fully functional applications.

The applications are Matrix Multiplication and K-Means Clustering. For a functional description of the app

- http://en.wikipedia
- http://en.wikipedia.org/wiki/K-means

The code optimization techniques you may want to explore include:

- OpenMP Pro??? (omp parallel, omp for. omp atomic, omp reduction, …)
- Cache blocking
- SIMD and Sequential Optimizations

Mini-Project 1

- Mini-Project Due — Monday, 3/8 @ 11:59PM EST
- Mini-Project One Review — Thursday, 3/11 @ 6:00PM EST

Handout: https://cmu.box.com/s/uzfkghtkzfs0 lu9o4

Gradescope Links:

- **Matrix Multiply:** https://www.gradescope.com/courses/241050/assignments/998372
- **K-Means:** https://www.gradescope.com/courses/241050/assignments/1025865
- **Project Writeup:** https://www.gradescope.com/courses/241050/assignments/997551

Mini-Project 1: gradescope

Mini-Project 1: gradescope

Mini-Project 1: gradescope

Mini-Project 1: gradescope

Mini-Project 1: gradescope

Mini-Project 1: gradescope

Mini-Project 1: Grading Criteria

Mini-Project 1: Write-Up

Mini-Project 1 - Setup

- **Matrix Multiple**

```
$ cd ~/18646_MP1/matrix_mul/omp
$ make
$ ./matrix_mul -i ../matrix_mul_03.dat -o
                ..
Test Case 10    11
```


- **K-Means**

```
$ cd ~/18646_MP1/kmeans
$ make
$ time ./omp_main -i ~/18646_MP1/data/kmeans03.dat -n 184
Writing coordinates of K=184 cluster centers…
Writing membership of N=191681 data objects…
115.659u 0.100s 0:15.57 743.4%
```

Mini-Project 1 – Makefile

- **The only two files you will submit are "`matrix_mul.cpp`" and "`omp_kmeans.c`"**
- **However!!!**
  - **For your REP ___ ent flags during compilation a**
  - `-mavx` (code changes may be
  - `-O3`

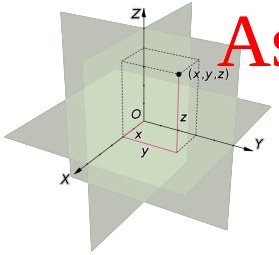
If you make changes to the **`Makefile`** for your fastest performing version, please include the code in your project report!!!

Outline

- CPU Achievable Peak Performance
- Matrix-Multiplication Discussion
- Overview of Mini-Project 1
- **A Quick Revie**
  - **SIMD Parallelis**
  - **OpenMP Pragmas**
  - **CPU Memory Hierarchy**
  - **False Sharing**
- Mini-Project 1.1 - Matrix Multiply
- Mini-Project 1.2 - K-Means

Exploiting SIMD Parallelism

- Applied to finding distance between two points:




```
V1 = _MM_LOAD(&X1[0])
V2 = _MM_LOAD(&Y1[0])
V1 = _MM_SUB(V2, V1)
V1 = _MM_MUL(V1, V1)
_MM_STORE(&res[0], V1)
add r1, &res[0], &res[1]
add r1, r1, &res[2]
sqrt r1, r1
st d, r1
```

```
ld r1, x1
ld r2, x2
ld r3, x3
ld r4, y1
ld r5, y2
ld r6, y3
sub r1, r4, r1
sub r2, r5, r2
sub r3, r6, r3
mul r1, r1, r1
mul r2, r2, r2
mul r3, r3, r3
add r1, r1, r2
add r1, r1, r3
sqrt r1, r1
st d, r1
```

SIMD Parallelism

```
#include <stdio.h>
#include <pmmintrin.h>  //  header for SSE3
#define k 32

int main(){
  float x[k]; float y[k];          // vectors of length k
  __m128 X, Y;                     // 128-bit values
  __m128 acc = _mm_setzero_ps(); // set to (0, 0, 0, 0)
  float inner_prod, temp
  int i, j;

  for (j=0; j<k; j++){ x

  for(i = 0; i < k - 4; i += 4) {
    X = _mm_load_ps(&x[i]);    // load chunk of 4
    Y = _mm_load_ps(y + i);    // alternate way, p
    acc = _mm_add_ps(acc, _mm_mul_ps(X, Y));
  }
  _mm_store_ps(&temp[0], acc);   // store acc into an array of floats
  inner_prod = temp[0] + temp[1] + temp[2] + temp[3];

  // add the remaining values
  for(; i < k; i++)
    inner_prod += x[i] * y[i];

  printf("Inner product is: %f\n", inner_prod);

  return 0;
}
```


SSE Version	gcc header
SSE	xmmintrin.h
SSE2	emmintrin.h
SSE3	Pmmintrin.h

Common SSE Intrinsics

```
__m128 _mm_setzero_ps(void)// set to (0, 0, 0, 0)
__m128 _mm_set_ps1(float f)// set to (f, f, f, f)
__m128 _mm_set_ps(float w, float x, float y , float z) // set to (z,y,x,w)
__m128 _mm_setr_ps(                                    ) // set to (w,x,y,z)
__m128 _mm_add_ps(_
__m128 _mm_sub_ps(__m128, __m128)
__m128 _mm_mul_ps(__m128, __m128)
__m128 _mm_load_ps(float *base_addr) // load 4 floats from base_addr
__m128 _mm_loadr_ps(float *base_addr) // load in reverse order void
_mm_store_ps(float *addr, __m128 val) // write val into location addr
```

CPU Memory Hierarchy

- **Cache**:
  - A piece of fast memory close to the compute modules in a microprocessor
  - Allows faster access to a limited amount of data
  - Physical properties of wires and transistors determines trade-off between **cache capacity**

ity, (throughput), (latency)

tes, 0.1 GB/s, ~1-10B cycles

6 GB/s, ~100 cycles

16GB, 34.08 GB/s, ~200 cycles

8 MB, 108.8 GB/s, 26-31 cycles

256 KB, 108.8 GB/s, 12 cycles

32 KB Data Cache, 163.2 GB/s, 4-6 cycles

Intel Core2 – 2600k @ 3.4GHz
(viewed from one core)

Working with a Memory Hierarchy

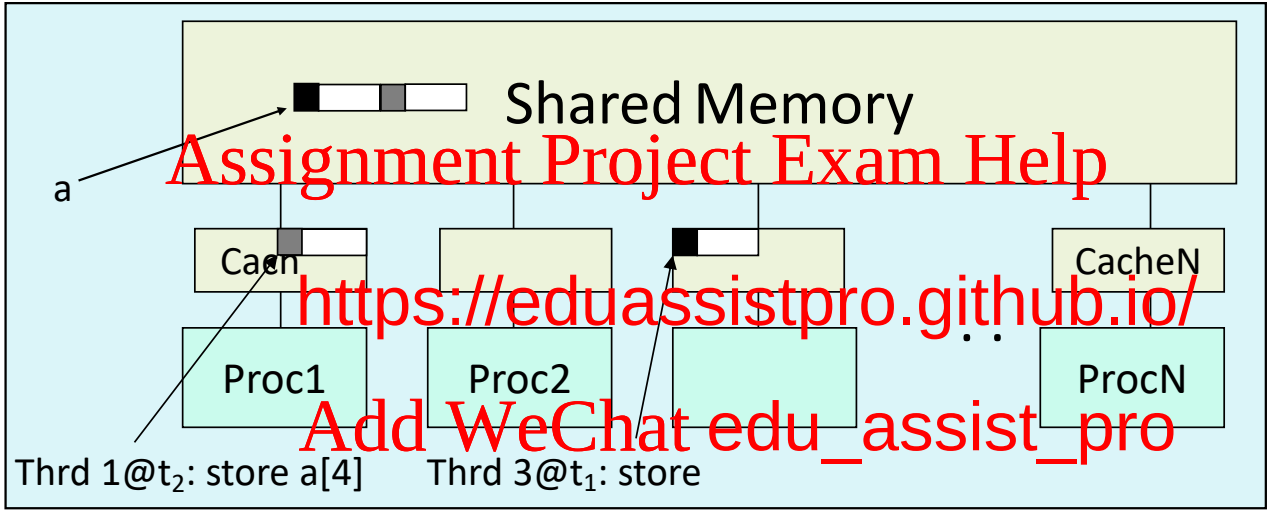
- **Principle of Locality**
  - The phenomenon of the same value or related storage locations being frequently accessed, usually amenable to performance optimization
- **Temporal Locality**
  - Reuse of specific and/or resources within relatively small time durations
- **Spatial Locality**
  - Use of data elements within relatively close storage locations

OpenMP Pragmas

- **OpenMP Pragmas**
  - for
  - atomic
  - reduction
  - private
  - shared
  - num_threads

- **OpenMP Scheduling**
  - schedule(static, dynamic,)

False Sharing



- Be aware of the cache line sizes for a platform
- Avoid accessing the same cache line from different threads

Outline

- CPU Achievable Peak Performance
- Matrix-Multiplication Discussion
- Overview of Mini-Project 1
- A Quick Revie
  - SIMD Parallelis
  - OpenMP Pragmas
  - CPU Memory Hierarchy
  - False Sharing
- **Mini-Project 1.1 - Matrix Multiply**
- Mini-Project 1.2 - K-Means

Matrix Multiply – Problem Criteria

Matrix Multiply: OpenMP

```
32 void
33 matrix_transpose_copy(float *matrix_dst, float *matrix_src, unsigned int dim)
34 {
35 #pragma omp parallel for
36     for (unsigned int i = 0; i < dim; i++) {
37       for (unsig
38         matrix_d                    (j * dim) + i];
39       }
40     }
41 }
```

Matrix Multiply: SIMD operations

```
65  __m128 vector_1, vector_2/*, vector_result*/;
67  for( unsigned int i = 0; i < sq_dimension; i++ ) {
68    for( unsigned int j = 0; j < sq_dimension; j++ ) {
69      //vector_result = _mm_setzero_ps();
70      sq_matrix_result[i*sq_dimension + j] = 0;
71      for( unsig                          =4 ) {
72         vector_1 = _mm_loadu_ps(&sq_matrix_1[i*sq_dimension_8 + k]);
73         vector_2 = _mm_loadu_ps(&sq_matrix_2[j*sq_dimension_8 + k]);
74         vector_1 = _mm_dp_ps(vector_                    );
75         sq_matrix_result[i*sq_dimens                    vtss_f32(vector_1);
77      }
78    }
79. }
```


- Could also:
  - Loop unrolling of inner loop to reduce overhead
  - Resize and pad original matrix to simplify code

Matrix Multiply: Cache blocking

```
48 #pragma omp parallel for private(i,j,k,i1,j1,k1)
49 for (i = 0; i < sq_dimension; i += B) {
50   for (j = 0; j < sq_dimension; j += B) {
51     for (k = 0; k < sq_dimension; k += B ) {
52       for (i1 =                                    i1++) {
53         mult_i =
54         for (j1 = j; j1 < j + B && j1 < sq_dimension; j1++) {
55           for (k1 = k; k1 < k + B && k                ; k1++) {
57             sq_matrix_result[mult_i +                 _1[mult_1 + k1] *
sq_matrix_2[k1*sq_dimension + j1];
58 }}}}}}
```


- Could also:
  - Unroll inner loops more to decrease overhead

Matrix Multiply: Transpose B

(probably don't need this for project 1)

```
32 void
33 matrix_transpose_copy(float *matrix_dst, float *matrix_src, unsigned int dim)
34 {
35 #pragma omp parallel for
36     for (unsigned int i = 0; i < dim; i++) {
37       for (unsi
38         matrix_                    M(          ,i,dim);
39       }
40     }
41 }
```


- Could also:
  - Use Cache Blocking for Transpose
  - Use SSE for Transpose

Outline

- CPU Achievable Peak Performance
- Matrix-Multiplication Discussion
- Overview of Mini-Project 1
- A Quick Revie
  - SIMD Parallelis
  - OpenMP Pragmas
  - CPU Memory Hierarchy
  - False Sharing
- Mini-Project 1.1 - Matrix Multiply
- **Mini-Project 1.2 - K-Means**

K-Means – Problem Criteria

K-means: The Phases

1. **Initialization:** Randomly select *k* cluster centers
   - Select *k* samples from data as initial centers [Forgy Partition]
2. **Expectation:** Assign each data point go closest center
   - Compare e r center (*k*)
   - Distance M nsions)
3. **Maximization:** Update centers nments
   - For each cluster (*k*) compute m ns) from data points assigned to that cluster
4. **Evaluate:** Re-iterate steps 2-3 until convergence or stopping criteria met
   - Percentage of data points re-assigned
   - Number of iterations (2-3)

Example Code (Expectation)

```
....
    delta = 0.0;
    for (i=0; i<numObjs; i++) {
      /* find the array index of nestest cluster center */
      index = find_near                                    [i], clusters);

          ...

      /* if membership changes, increase delta by 1 */
      if (membership[i] != index) delta += 1.0;

      /* assign the membership to object i */
      membership[i] = index;
...
```


Evaluate distance to each cluster centroid and select closest

Example Code (Expectation)

```
....
    delta = 0.0;
    for (i=0; i<numObjs; i++) {
      /* find the array index of nestest cluster center */
      index = find_near                                    [i], clusters);
```


Active Data Structur


```
    membership[i] = index;
...
```

Possible Concurrencies:

- N (independent)
- D (sum reduction) ← euclid_dist_2()
- k (min reduction)

Example Code (Maximization)

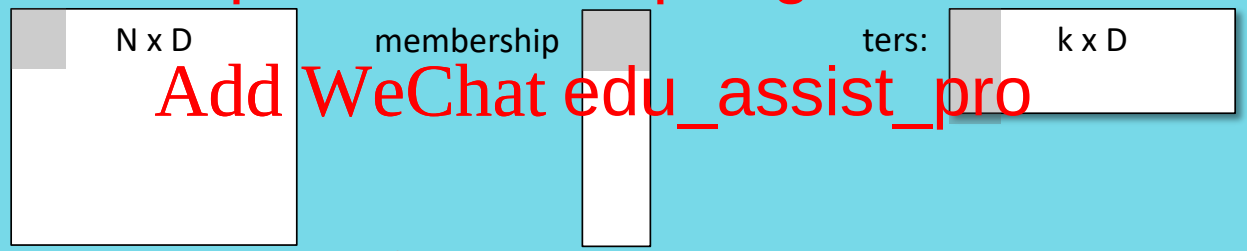
```
....
    /* update new cluster centers : sum of objects located within */
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            newClusters[ind
    }

    /* average the sum and replace old cluster cente                    */
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            if (newClusterSize[i] > 0)
                clusters[i][j] = newClusters[i][j] / newClusterSize[i];
            newClusters[i][j] = 0.0;   /* set back to 0 */
        }
        newClusterSize[i] = 0;       /* set back to 0 */
    }
...
```

prepare for next iteration

Example Code (Maximization)

```
....
    /* update new cluster centers : sum of objects located within */
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
          newClusters[ind
```


Active Data Structur




```
        }
        newClusterSize[i] = 0;      /* set bac
    }
...
```

Possible Concurrencies:

D (independent)

N (Histogram computation into k bins

K-Means: Parallelization

```
#pragma omp parallel for
  private(i, ???)
  firstprivate(numObjs, ???)
  shared(objects, ???)
  reduction(???)
```


- How can we
- Could we introduce additional l n each thread?
- Can we parallelize this further?

K-Means: Cost Reduction

```
for (i=0; i<numCluster; i++){
        float tmp = 1.0f / newClusterSize[i];
        for (j = 0; j < numCoords; j++)
            if (newClusterSize[i] > 1)
                clusters[i][j] = newClusters[i][j] * tmp;
}
```


- Remove the dividing operation a uter loop

K-Means: Other Ideas?

- Predict "stable" points, which means that if a point doesn't change its cluster for "m" rounds, we regard it as "stable" and skip the computation next round