# CS 377P Assignment 4 Help Session

TA: Ruei-Bang Chen (slides adapted from Yi-Shan Lu)

CS, UT Austin

3/27/2019

# Outline

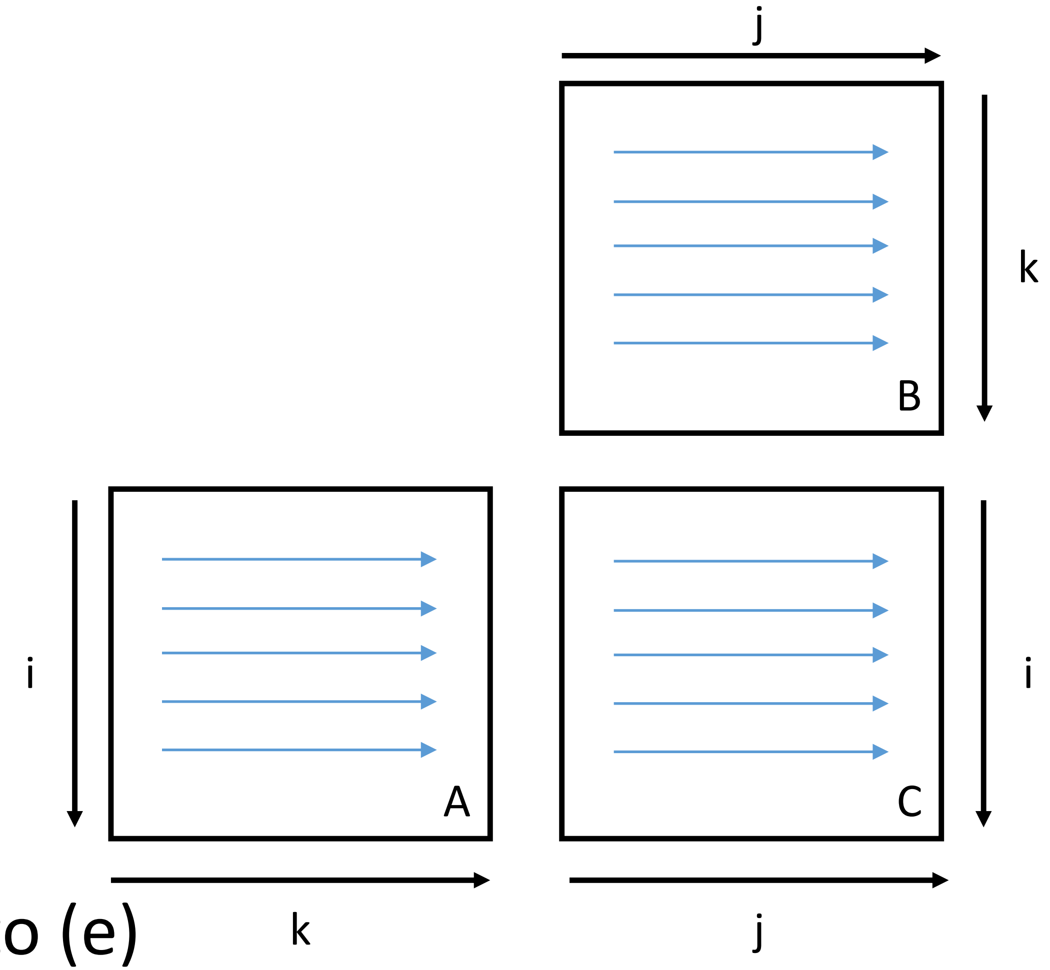
- Guide for subproblems
- Notes on measurement
- Implementation tricks

# Guides for Subproblems

# MMM Loop Nests

```
for (i = 0; i < sz; i++) {
  for (k = 0; k < sz; k++) {
    for (j = 0; j < sz; j++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

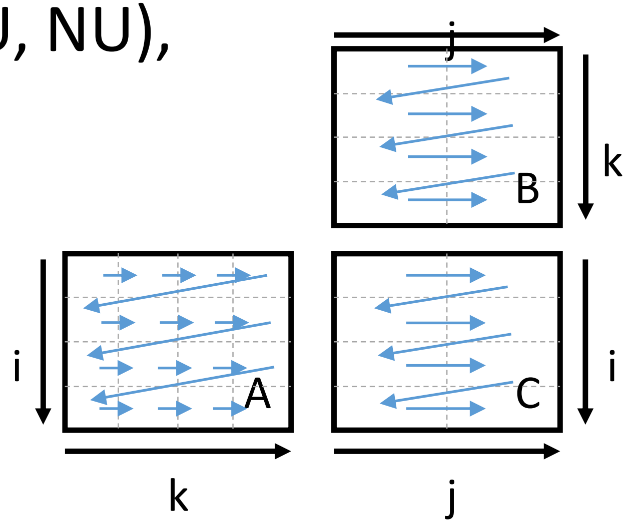
You can use other loop ordering,
but be consistent across part (a) to (e)



# Micro-kernel: Register Tiling

```
//mini-kernel
for(int j = 0; j < NB; j += NU)
  for (int i = 0; i < NB; i += MU)
    load C[i..i+MU-1, j..j+NU-1] into registers
      for (int k = 0; k < NB; k++)
        //micro-kernel
        load A[i..i+MU-1,k] into registers
        load B[k,j..j+NU-1] into registers
        multiply A's and B's and add to C's
        store C[i..i+MU-1, j..j+NU-1]
```

- Be aware of the loop ordering.
- You can use MU and NU values from the Yotov paper.
  - They suggest MU = 5 or 6, NU = 1 for JIK loop nests
  - But feel free to use other values as long as they make sense
  - Note that for the next part you have to use a multiple of 4 due to the vectorization
- To avoid cleanup code, matrix size N = c*LCM(MU, NU), where c is an integer
- Allocate registers in a portable way.
  - register type var = array[index];
- NB = N for now.
  - Mini-kernel = full MMM in this case.



# Vectorization

- Sufficient to replace/merge scalar registers with vector registers.
- See https://software.intel.com/sites/landingpage/IntrinsicsGuide/ for the available vector intrinsic functions.
- See examples of using SSE/SSE2 intrinsic functions at https://www.cs.fsu.edu/~engelen/courses/HPC-adv/MMXandSSEexamples.txt
- Note that we use float in this assignment

# Example of Using Vector Intrinsics

```
float A[size], B[size], C[size];

// assume that size is a multiple of 4
void vec_float_add(float* c, float* a, float* b) {
  for (int i = 0; i < size; i += 4) {
    __m128 vec_a = _mm_load_ps(a+i);
    __m128 vec_b = _mm_load_ps(b+i);
    _mm_store_ps(c+i, _mm_add_ps(vec_a, vec_b));
  }
}

void some_func() {
  ...
  vec_float_add(C, A, B);
  ...
}
```

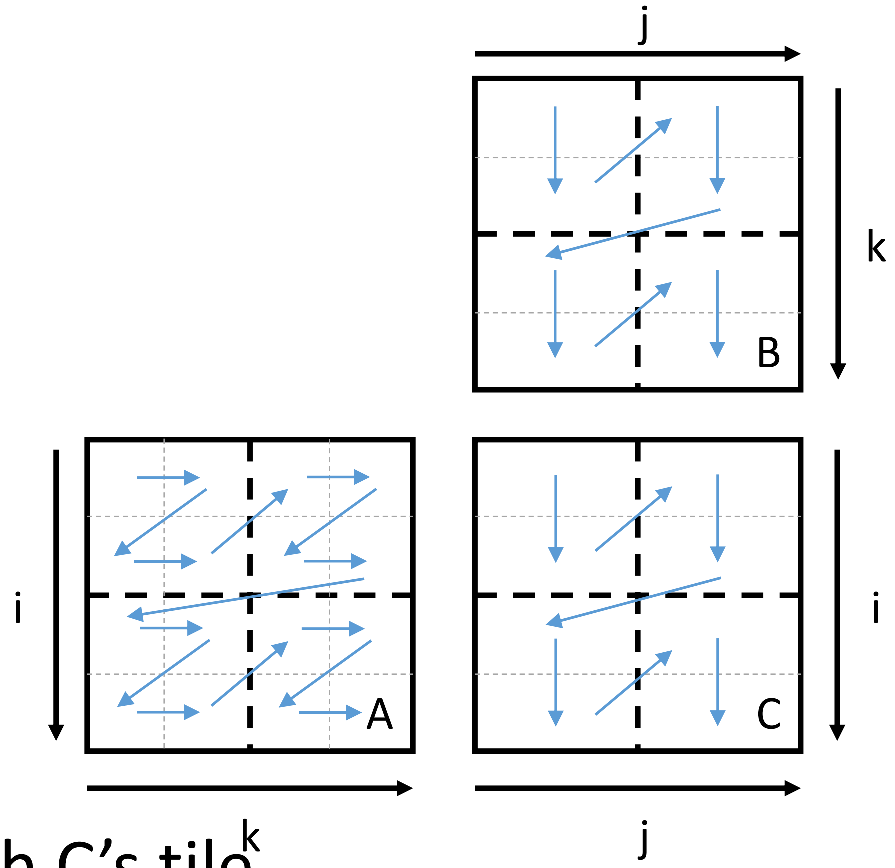
The vector counterpart of a scalar register

# Mini-kernel: L1 Cache Tiling

- To avoid cleanup code,
  - NB = c * LCM(MU, NU).
  - Matrix size N = c’ * NB, where c’ is an integer.
- Micro-kernel works inside mini-kernel, which processes tiles of NB by NB, NB <= N.
- Experiment with different NB and pick the one that works best
- Add 3 loops outside of the mini-kernel to have a full MMM.
  - These loops control which tiles are used for computation.

# Buffering the Tiles

- Key questions:
  - Which matrix needs only one element;
  - Which matrix needs only one row/column;
  - Which matrix needs to be fully in L1 cache; and
  - When to copy a tile in to/out from a buffer.
- Figure out the above from the loop ordering
- Copy back to the original C after finishing with C's tile.
- Use memcopy for the copying



# MKL

- Example https://software.intel.com/en-us/mkl-tutorial-c-multiplying-matrices-using-dgemm#9CEED00C-1A85-4AC0-8AF8-BE2AFEF0E603
  - Note that the example uses double type
  - Use cblas_sgemm instead of cblas_dgemm for float type
  - https://software.intel.com/en-us/mkl-developer-reference-c-cblas-gemm

- The trend for GFLOPS might be different
  - Think about how GFLOPS is calculated
  - Pay careful attention to your raw measurement values, especially total floating point operations
  - Figure out an explanation
  - Assume the number of floating point operations as $2n^3$
  - Divide it by the measured running time to get FLOPS

# Notes on Measurement

# Do Remember to (Lesson from Assignment 1)

- Flush all three levels of data caches.
  - Get the same initial state across different runs.
  - Allocate a large enough array, and walk through it to evict everything else.
- Use serializing instructions right before and right after the measured code.
  - To avoid compiler optimization and hardware out-of-order execution.
  - Example: __cupid() in <cupid.h>, see https://en.wikipedia.org/wiki/CPUID

# Performance

- FLOPS = Floating-point Operations Per Second

  - Need to measure the absolute runtime and the number of total floating point operations

  - Be careful when calculating the total number of floating point operations for vectorized code as stated in the next slide

# Validating Your Measurement

- Use PAPI_FP_OPS for this purpose.
- For the same size of matrices, part (a) to (e) of your code should have roughly the same number of floating-point operations.
  - Part (a) & (b): PAPI_FP_OPS
  - Part (c), (d) & (e): vector_width * PAPI_FP_OPS
    - We are counting # double/single-precision operations, but PAPI_FP_OPS reports # hardware operations.
  - vector_width: 2 for double-precision FP, 4 for single-precision FP (128 bits in total)
    - No AVX on the orcrists

# Implementation Tricks

# Navigating a Large Configuration Space

- Parameterize your program so it is easier to try different configurations through command-line arguments.
  - Matrix size
  - Tiling mode: five subproblems
  - Measurement mode: runtime, PAPI events, etc.
- Build your code for different versions
  - Makefile for compilation with make
  - #ifdef, #if, etc. in your source to have conditional compilation (via C preprocessor, CPP)
- Use a (bash) script to iterate over configurations.
- Write or redirect your program output to files for post-processing.
- Use gcc and –O2 for part (a) to (e), you can separate part (f) from others

# Useful Command-line Utilities

- Simplification of the I/O processing for your program
  - Input redirection: <
  - Output redirection: >, &>, etc.
- Comparison & correctness verification: diff / vimdiff
- Show file contents: head, tail, cat, etc.
- String/file manipulation: sed/awk, join, fgrep, sort, etc.