CS 377P Assignment 3
Help Session

TA: Yi-Shan Lu
CS, UT Austin

3/2/2018
Outline

• Guide for subproblems
• Notes on measurement
• Implementation tricks
Guides for Subproblems
MMM w/ IKJ Loop Nests

```c
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];
        }
    }
}
```
Micro-kernel: Register Tiling

• Be aware of the loop ordering.
  • IKJ in this assignment.

• You can use MU and NU values from the Yotov paper.
  • MU = 5 or 6, NU = 1 for JIK loop nests.

• To avoid cleanup code, matrix size $N = c \times \text{LCM}(MU, NU)$.

• 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 examples of using SSE/SSE2 intrinsic functions at https://www.cs.fsu.edu/~engelen/courses/HPC-adv/MMXandSSEexamples.txt
Example of Using Vector Intrinsics

```c
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.

• Micro-kernel works inside mini-kernel, which processes tiles of NB by NB, NB <= N.

• 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 (IKJ for this assignment).
• Copy back to the original C after finishing with C’s tile.
Notes on Measurement
Peak Performance

• FLOPS = FLoating-point Operations Per Second
  • Need to measure absolute runtime.

• 9.6 G DP FLOPS for a single core of Intel Xeon E5530 CPUs on the orcrists.
  • 4 double-precision (DP) floating point operations (FLOPs) per cycle.
    • 2 DP multiplications.
    • 2 DP additions.
  • Highest frequency: 2.4 GHz.
  • 4 * 2.4G = 9.6G
Do Remember to...

• 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](https://en.wikipedia.org/wiki/CPUID)
Validating Your Measurement

• Use PAPI_FP_OPS for this purpose.

• For the same size of matrices, all five variants 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
    • 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.
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.