TD TECH TALKS

LOW-LEVEL PERFORMANCE OPTIMIZATION: CACHE LOCALITY AND OPENMP

ARTEM FROLOV

Quantitative development, UQL

HIGH LEVEL PERFORMANCE OPTIMIZATION

(Largely CPU and language independent)

- Algorithms
- Data structures
- Asynchronous I/O
- Binary serialization
- Database indices
- Task parallelization

LOW LEVEL PERFORMANCE OPTIMIZATION

(With understanding of CPU, memory, and compilation)

- Cache locality
- CPU pipeline
- Vectorization
- Data parallelization

THIS TALK

- Cache locality
- Parallelization with OpenMP

HIGH-LEVEL VS. LOW-LEVEL

- Most code (esp. user facing) is I/O bound
- Low-level makes most sense for CPU-bound and memory-bound code
  - Quantitative libraries
  - Scientific and technical computing
  - System-level code (drivers, DBs, graphical libraries,...)
- Do low-level opt only after high-level opt

LATENCY NUMBERS EVERY PROGRAMMER SHOULD KNOW

operation	time (ns)	time (ms)	comment
L1 cache reference	0.5 ns		
Branch mispredict	5 ns		
L2 cache reference	7 ns		
Mutex lock/unlock	25 ns		
RAM access	100 ns		
Compress 1KB (Zippy)	3000 ns		

LATENCY NUMBERS (2/3)

operation	time (ns)	time (ms)	comment
Send 1K bytes over 1Gbps network	10,000 ns	0.01 ms	
Read 4K randomly from SSD	150,000 ns	0.15 ms	
Read 1MB randomly from memory	250,000 ns	0.25 ms	
Round trip within datacenter	500,000 ns	0.5 ms	

LATENCY NUMBERS (3/3)

operation	time (ns)	time (ms)	comment
Read 1 MB sequentially from SSD	1,000,000 ns	1 ms	
Disk seek	10,000,000 ns	10 ms	20X datacenter
Read 1MB sequentially from disk	20,000,000 ns	20 ms	80X RAM, 20X SSD
Send packet CA, US -> NL -> CA, US	150,000,000 ns	150 ms	

CACHE LOCALITY

- Memory is cached by chunks (cache lines)
- Accesses better be bunched together in time and space
- Arrays: successive > random, smaller stride
- Data structures: less pointers
- Lists and hash tables: need chunking optimization
- Temporal locality: recently referenced are more likely to be referenced in future
- Spatial locality: items closely together tend ti be referenced close together in time

DATA LAYOUT

- Matrices: row-major vs column-major
- Array of structures (AOS)
- Structure of arrays (SOA)

ROW-MAJOR VS. COLUMN-MAJOR

MATRIX

11	12	13	14
21	22	23	24
31	32	33	34

ROW-MAJOR

11	12	13	14	21	22	23	24	31	32	33	34

COLUMN-MAJOR

11	21	31	12	22	32	13	23	33	14	24	34

TESTING PEARSON CORRELATION

```
inline size_t matrix_idx(size_t n_row_size,
    size_t n_column_size, size_t i, size_t j) {
#if MATRIX_ORDER_ROW_MAJOR
    return i * n_row_size + j;
#else
    return j * n_column_size + i;
#endif
}
inline double matrix_get(double* mtx, size_t n_row_sz,
    size_t n_col_sz, size_t i, size_t j) {
    return mtx[matrix_idx(matrix, n_row_sz, n_col_sz, i, j)];
}
inline void matrix_set(double* mtx, size_t n_row_sz,
    size_t n_col_sz, size_t i, size_t j, double val) {
    mtx[matrix_idx(matrix, n_row_sz, n_col_sz, i, j)] = val;
}
```

CACHE LOCALITY: DEMO

PEARSON CORRELATION RESULTS

Series number and length	Row-major time	Column-major time	Factor
256 x 256	0.152	0.172	1.13x
512 x 512	1.242	1.486	1.19x
1024 x 1024	10.716	40.331	3.76x
2048 x 2048	80.094	368.268	4.82x

ARRAY OF STRUCTURES AND STRUCTURE OF ARRAYS

```
struct pixel_t {
    uint8_t r;
    uint8_t g;
    uint8_t b;
} ArrayOfStructures[N];

struct struct_of_arrays_t {
    uint8_t r[N];
    uint8_t g[N];
    uint8_t b[N];
} StructureOfArrays;
```

OPENMP

- Fine-grained parallelization
- API to support shared memory multiprocessing
- C, C++, and Fortran
- Set of compiler pragmas

OPENMP: HOW IT WORKS

- For each section *master* thread *forks* a number of *slave* threads
- Work sharing constructs specify division of work ans synchronization
- At the end of section *slave* threads *join* into the master thread

OPENMP: SIMPLEST EXAMPLE

```
float a[8192], b[8192], c[8192];
#pragma omp for
for (i=0; i<sizeof(a)/sizeof(a[0]); i++)
    c[i] = a[i] + b[i];
```

OPENMP: THREAD-LOCAL STORAGE, SCHEDULING, AND CRITICAL SECTIONS

```
float a[8192], b[8192], c[8192];
double sum = 0.0, loc_sum = 0.0;
#pragma omp parallel private(loc_sum)
{
    #pragma omp for(static, 1)
    for (i=0; i<sizeof(a)/sizeof(a[0]); i++) {
        c[i] = a[i] * b[i];
           loc_sum += c[i];
    }
    #pragma omp critical
    sum = sum + loc_sum;
}
```

OPENMP: SUPPORT FOR REDUCTION

```
float a[8192], b[8192], c[8192];
long sum = 0;
 #pragma omp parallel for reduction(+:sum) schedule(static,1)
 for(i = 0; i < N; i++) {
     sum = sum + a[i]*b[i];
 }
```

OPENMP: DYNAMIC SCHEDULING

```
#pragma omp parallel private(j,k)
{
  #pragma omp for schedule(dynamic, 1)
  for(i = 2; i <= N-1; i++)
     for(j = 2; j <= i; j++)
        for(k = 1; k <= M; k++)
           b[i][j] += a[i-1][j]/k + a[i+1][j]/k;
}
```

OPENMP: DEMO

OPENMP

- pros
  - Simple
  - Portable
  - Piece-meal approach to parallelization
- cons
  - Risk of race conditions
  - Maybe difficult to debug
  - Error-handling is compilcated

THANK YOU!

Q&A

Artem.Frolov@tdsecurities.com