CS 110
Computer Architecture

Amdahl's Law, Data-level Parallelism (Auxiliary Slides)

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/22s/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkeley's CS61C

DGEMM Speed Comparison

- Double precision GEneral Matrix Multiply: DGEMM
- Intel Core i7-5557U CPU @ 3.10 GHz
  - Instructions per clock (mul_pd) 2; Parallel multiplies per instruction 4
  - => 24.8 GFLOPS
- Python:

```
def dgemm(N, a, b, c):
    for i in range(N):
        for j in range(N):
            c[i+j*N] = 0
            for k in range(N):
                c[i+j*N] += a[i+k*N] * b[k+j*N]
```

N	Python [Mflops]
32	5.4
160	5.5
480	5.4
960	5.3

- 1 MFLOP = 1 Million floating-point operations per second (fadd, fmul)
- **dgemm**(N …) takes $2*N^3$ flops

C versus Python

- c = a * b
- a, b, c are N x N matrices

```
// Scalar;  P&H p. 226
void dgemm_scalar(int N, double *a, double *b, double *c) {
    for (int i=0;  i<N;  i++)
        for (int j=0;  j<N;  j++) {
            double cij = 0;
            for (int k=0;  k<N;  k++)
                //     a[i][k]  * b[k][j]
                cij += a[i+k*N] * b[k+j*N];
            // c[i][j]
            c[i+j*N] = cij;
        }
}
```

N	C [GFLOPS]	Python [GFLOPS]
32	1.30	0.0054
160	1.30	0.0055
480	1.32	0.0054
960	0.91	0.0053

Vectorized dgemm

```
// AVX intrinsics;  P&H p. 227
void dgemm_avx(int N, double *a, double *b, double *c) {
    // avx operates on 4 doubles in parallel
    for (int i=0;  i<N;  i+=4) {
        for (int j=0;  j<N;  j++) {
            // c0 = c[i][j]
            __m256d c0 = {0,0,0,0};
            for (int k=0;  k<N;  k++) {
                c0 = _mm256_add_pd(
                        c0,   // c0 += a[i][k] * b[k][j]
                        _mm256_mul_pd(
                            _mm256_load_pd(a+i+k*N),
                            _mm256_broadcast_sd(b+k+j*N)));
            }
            _mm256_store_pd(c+i+j*N, c0); // c[i,j] = c0
        }
    }
}
```

N	Gflops	
	scalar	avx
32	1.30	4.56
160	1.30	5.47
480	1.32	5.27
960	0.91	3.64

- 4x faster
- Still << theoretical 25 GFLOPS

Loop Unrolling

```
// Loop unrolling;  P&H p. 352
const int UNROLL = 4;

void dgemm_unroll(int n, double *A, double *B, double *C) {
    for (int i=0;  i<n;  i+= UNROLL*4) {
        for (int j=0;  j<n;  j++) {
            __m256d c[4];
            for (int x=0;  x<UNROLL;  x++)
                c[x] = _mm256_load_pd(C+i+x*4+j*n);
            for (int k=0;  k<n;  k++) {
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                for (int x=0;  x<UNROLL;  x++)
                    c[x] = _mm256_add_pd(c[x],
                                _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
            }
            for (int x=0;  x<UNROLL;  x++)
                _mm256_store_pd(C+i+x*4+j*n, c[x]);
        }
    }
}
```

4 registers

Compiler does the unrolling

N	GFlops		
	scalar	avx	unroll
32	1.30	4.56	12.95
160	1.30	5.47	19.70
480	1.32	5.27	14.50
960	0.91	3.64	6.91 ?

FPU versus Memory Access

- How many floating-point operations does matrix multiply take?
  - $F = 2 \times N^3$ (N^3 multiplies, N^3 adds)
- How many memory load/stores?
  - $M = 3 \times N^2$ (for A, B, C)
- Many more floating-point operations than memory accesses
  - $q = F/M = 2/3 * N$
  - Good, since arithmetic is faster than memory access
  - Let's check the code …

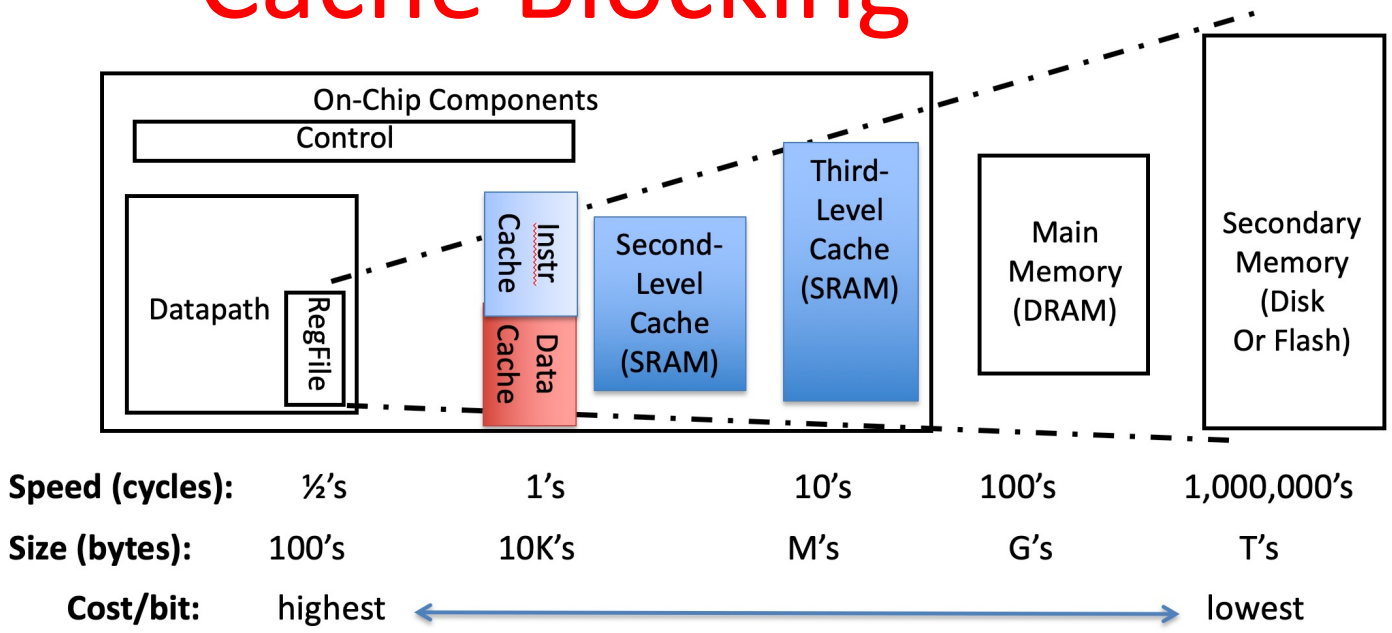
But memory is accessed repeatedly

- q = F/M = 1.6! (1.25 loads and 2 floating-point operations)

Inner loop:

```
for (int k=0;  k<N;  k++) {
    c0 = _mm256_add_pd(
            c0,   // c0 += a[i][k] * b[k][j]
            _mm256_mul_pd(
                _mm256_load_pd(a+i+k*N),
                _mm256_broadcast_sd(b+k+j*N)));
}
```

Cache Blocking

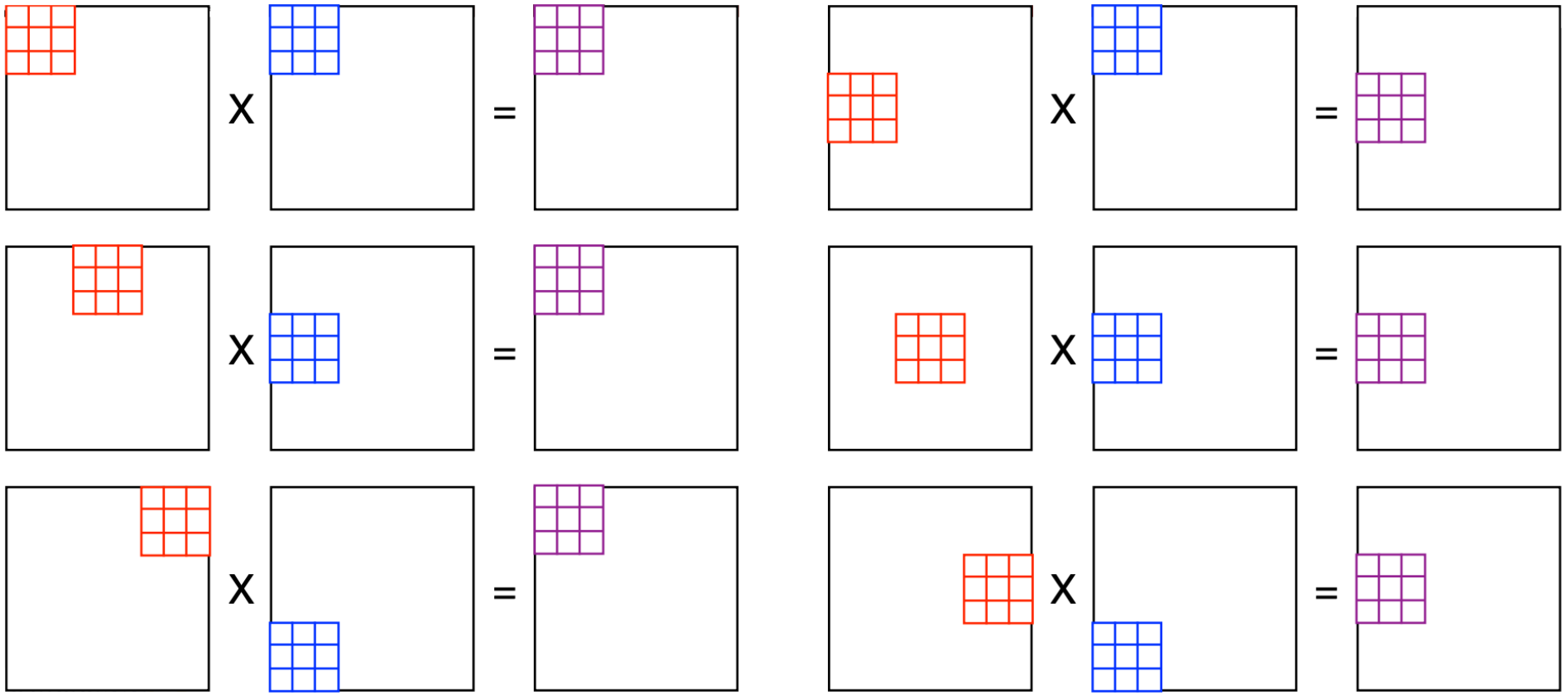


- Where are the operands (A, B, C) stored?
- What happens as N increases?
- Idea: arrange that most accesses are to fast cache!
- Rearrange code to use values loaded in cache many times
- Only "few" accesses to slow main memory (DRAM) per floating point operation

P&H, RISC-V edition p. 465

  - -> throughput limited by FP hardware and cache, not slow DRAM

Blocking Matrix Multiply
(divide and conquer: sub-matrix multiplication)



Memory Access Blocking

```
// Cache blocking;  P&H p. 556
const int BLOCKSIZE = 32;

void do_block(int n, int si, int sj, int sk, double *A, double *B, double *C) {
    for (int i=si;  i<si+BLOCKSIZE;  i+=UNROLL*4)
        for (int j=sj;  j<sj+BLOCKSIZE;  j++) {
            __m256d c[4];
            for (int x=0;  x<UNROLL;  x++)
                c[x] = _mm256_load_pd(C+i+x*4+j*n);
            for (int k=sk;  k<sk+BLOCKSIZE;  k++) {
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                for (int x=0;  x<UNROLL;  x++)
                    c[x] = _mm256_add_pd(c[x],
                                _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
            }
            for (int x=0;  x<UNROLL;  x++)
                _mm256_store_pd(C+i+x*4+j*n, c[x]);
        }
}

void dgemm_block(int n, double* A, double* B, double* C) {
    for(int sj=0;  sj<n;  sj+=BLOCKSIZE)
        for(int si=0;  si<n;  si+=BLOCKSIZE)
            for (int sk=0;  sk<n;  sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
```

Performance

- Intel i7-5557U theoretical limit (AVX2): 24.8 GFLOPS
- Cache:
  - L3: 4 MB 16-way set associative shared cache
  - L2: 2 x 256 KB 8-way set associative caches
  - L1 Cache: 2 x 32KB 8-way set associative caches (2x: D & I)
- Maximum memory bandwidth (GB/s): 29.9

N	Size	GFlops			
		scalar	avx	unroll	blocking
32	3x 8KiB	1.30	4.56	12.95	13.80
160	3x 200KiB	1.30	5.47	19.70	21.79
480	3x 1.8MiB	1.32	5.27	14.50	20.17
960	3x 7.2MiB	0.91	3.64	6.91	15.82

Intel Math Kernel Library

- AVX programming too hard? Use MKL!
  - C/C++ and Fortran for Windows, Linux, macOS
- Knowledge about AVX still very helpful for using MKL (e.g. Cache blocking, …)
- MKL also for multi-threading…

DGEMM, SGEMM Optimized by Intel® Math Kernel Library on Intel® Xeon® Processor