Advanced Topics in Numerical Analysis: High Performance Computing

More Shared Memory Parallelism, Sequential Performance Demo, Libraries

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

March 11, 2019

Outline

Organization

Scheduling:

- Homework assignment #2 due tonight
- New assignment will be posted this week

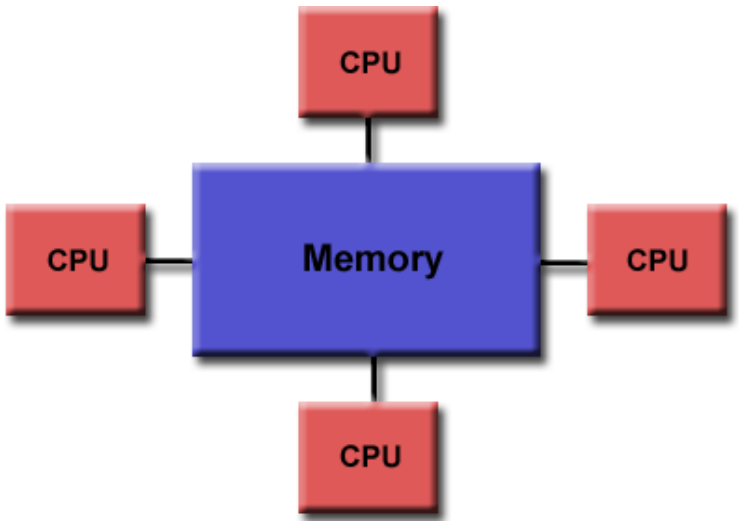
Topics today:

- Wrapping up shared memory parallelism
- Revisit sequential performance (finish demo from lecture 3)
- Libraries BLAS, LAPACK, FFTW

Outline

Shared memory programming model

- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition



Only one process is running, which can fork into shared memory threads.

#pragma omp for schedule (static/dynamic/guided [,chunk])

- **static** divided into pieces of size chunk, and statically assigned to threads.
- **dynamic** divided into pieces of size chunk and dynamically scheduled as requested.
- **guided** size of chunk decreases over time.

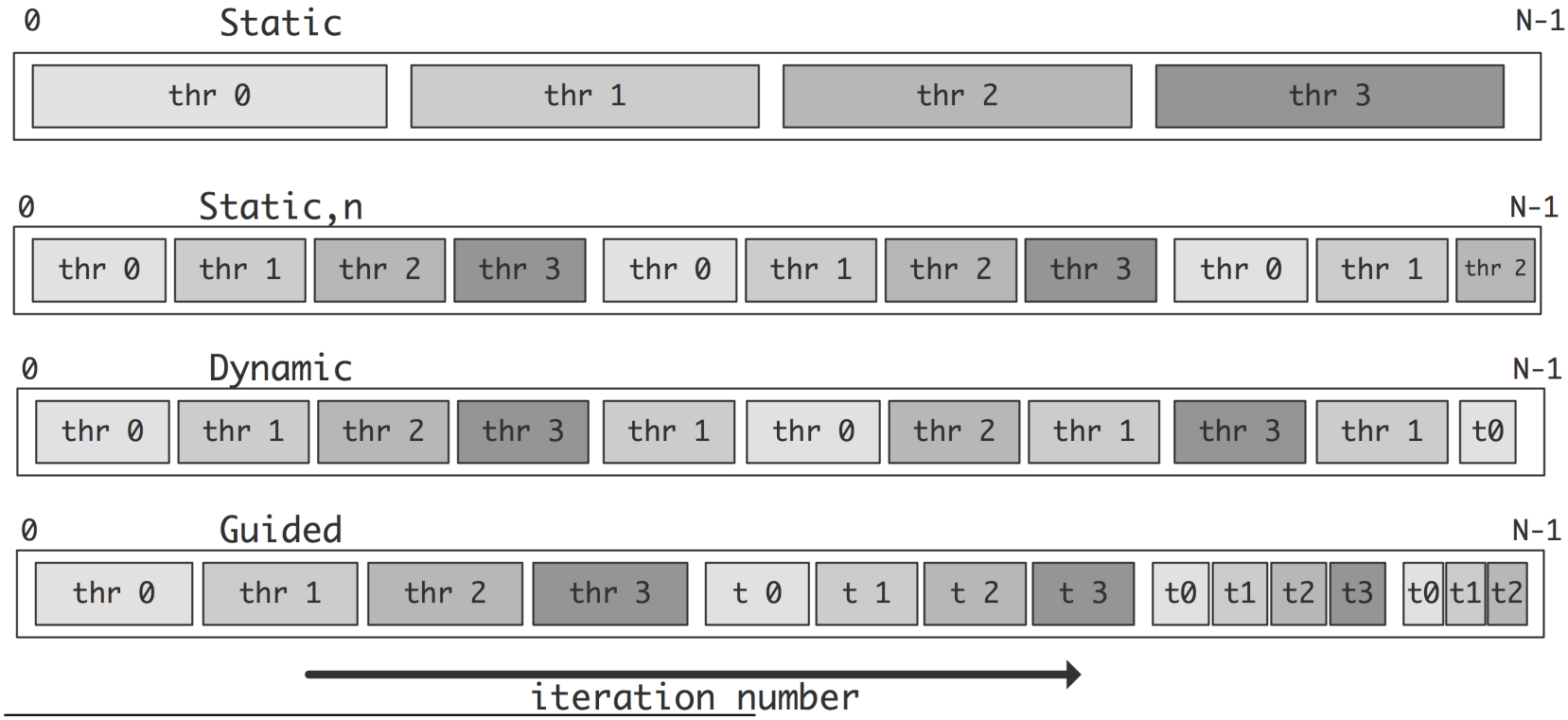


*figure from http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Weak sequential consistency

```
int a = 0, b = 0;
```

Thread-1

```
a = 1;
b = 2;
```

Thread-2

```
if (b == 2)
  assert(a == 1);
```

(may fail because b may be commited before a)

Weak sequential consistency

```
int a = 0, b = 0;
```

Thread-1

```
a = 1;
#pragma omp flush(a)
b = 2;
```

Thread-2

```
if (b == 2)
  assert(a == 1);
```

Synchronization

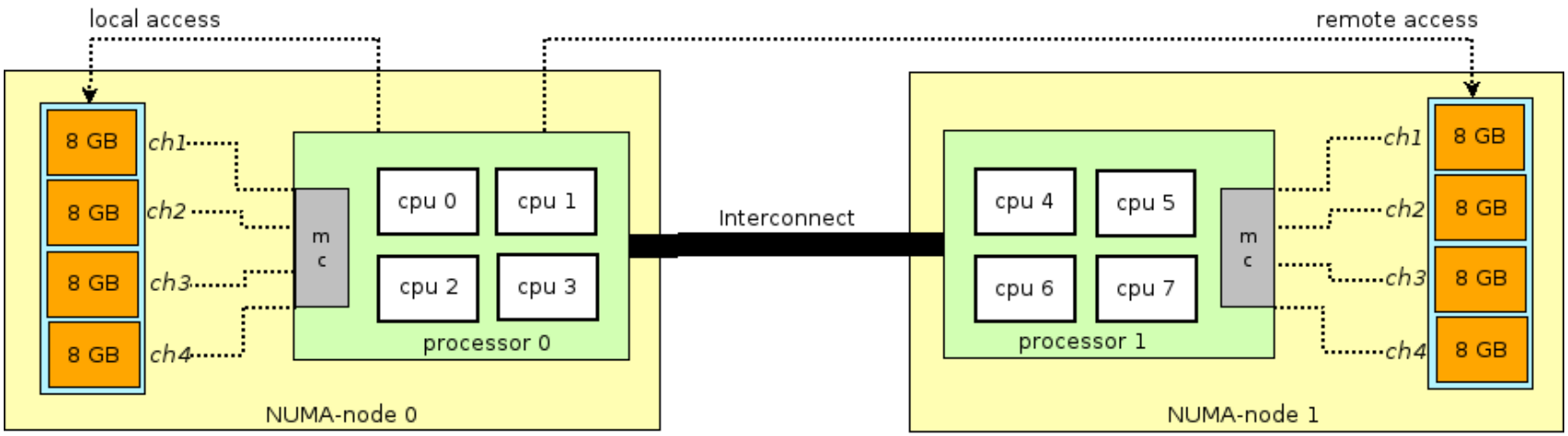
Synchronization: needed to protect access to shared data

- Implicit barrier synchronization at end of parallel region (noexplicit support for synch. subset of threads). Can invoke explicitly with `#pragma omp barrier`. All threads must see same sequence of work-sharing and barrier regions.
- critical sections: only one thread at a time in critical region with the same name. `#pragma omp critical [(name)]`
- atomic operation: protects updates to individual memory loc. Only simple expressions allowed. `#pragma omp atomic`
- also flush, locks, master operations

At all these (implicit or explicit) synchronization points OpenMP ensures that threads have consistent values of shared data.

Cache Coherent Non-uniform Memory Access

- **Cores:** individual processing units.
- **Sockets:** collection of cores on the same silicon die.
- Each sockets connected to its own DRAM.
- Sockets interconnected using a network: QPI (Intel), HT (AMD).
- Location of memory pages determined by first-touch policy.

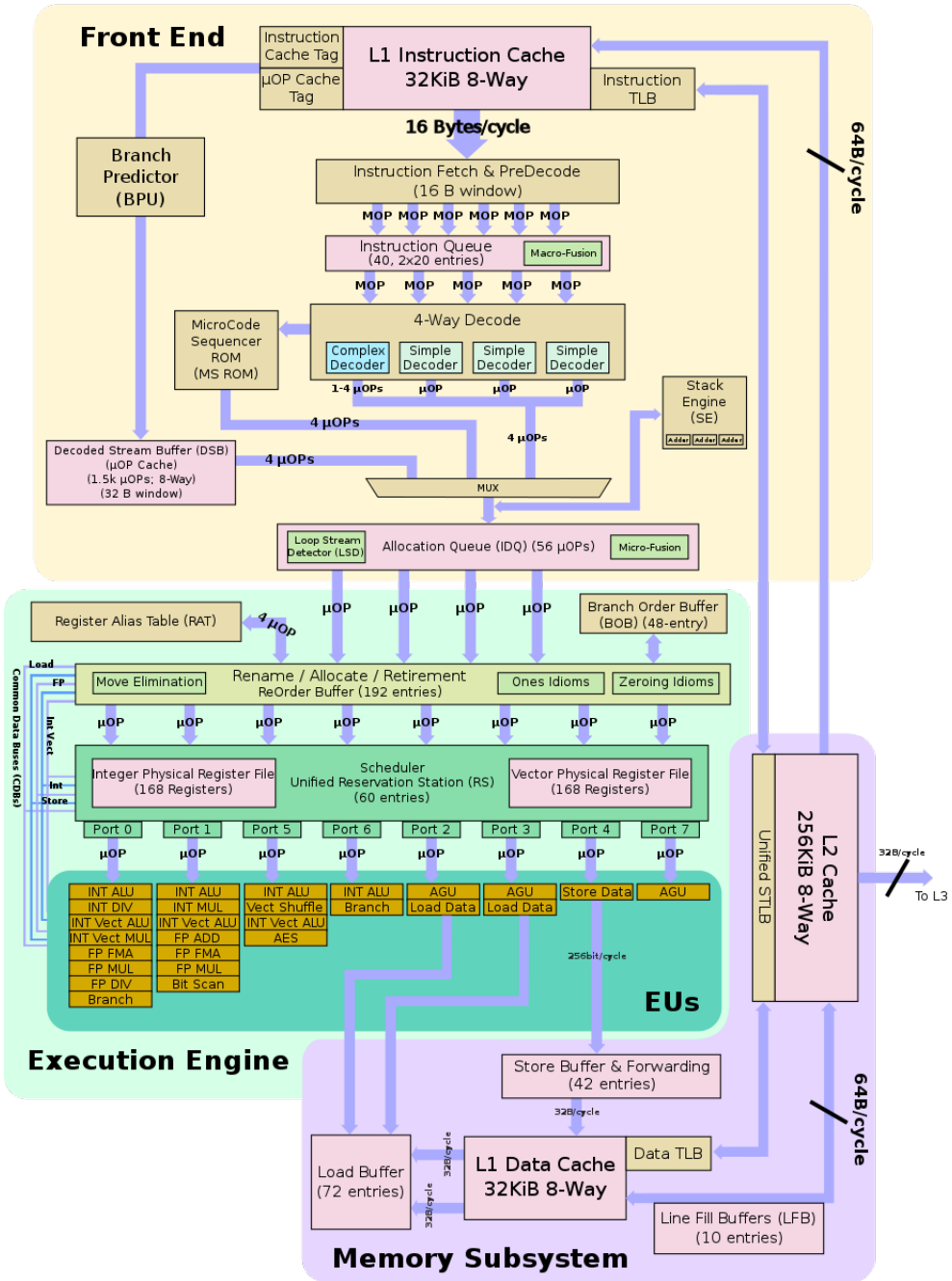


†

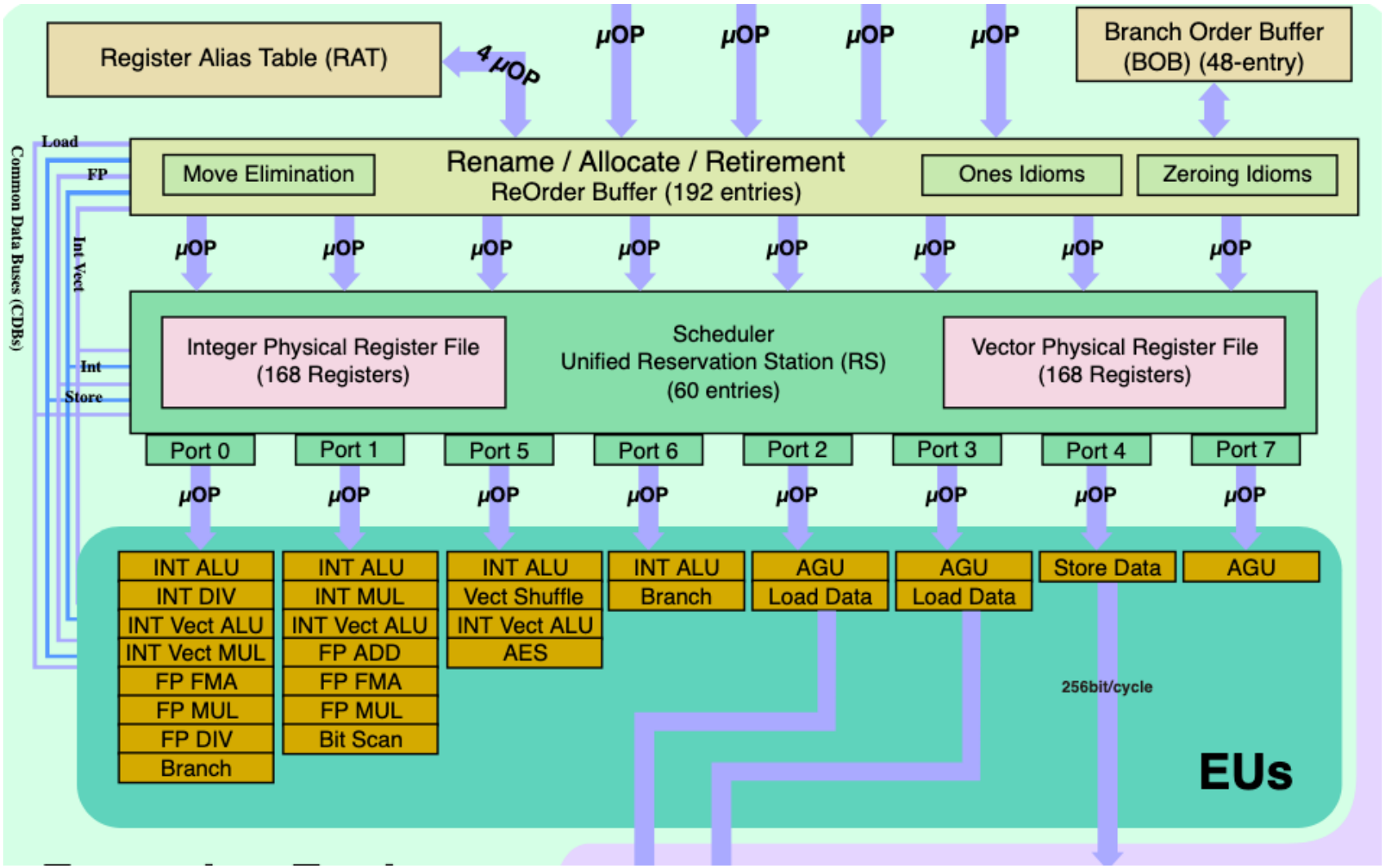
†figure from: https://www.boost.org

Outline

Sequential Performance



Sequential Performance



Sequential Performance

Instruction Level Parallelism (ILP)

- **Out of order execution** re-order instructions
- **Pipelining** "assembly line" parallelism
- **Superscalar architecture** multiple execution units
- **Branch prediction** speculative execution

Compilers and processors already very good at this part!
How can we help?

Sequential Performance

Instruction Level Parallelism (ILP)

- **Out of order execution** re-order instructions
- **Pipelining** "assembly line" parallelism
- **Superscalar architecture** multiple execution units
- **Branch prediction** speculative execution

Compilers and processors already very good at this part!
How can we help? Keep data close to processor to avoid pipeline stalls, loop unrolling to expose independent instructions, avoid instruction dependencies, avoid conditional branches etc.

Sequential Performance

Instruction Level Parallelism (ILP)

- **Out of order execution** re-order instructions
- **Pipelining** "assembly line" parallelism
- **Superscalar architecture** multiple execution units
- **Branch prediction** speculative execution

Compilers and processors already very good at this part!
How can we help? Keep data close to processor to avoid pipeline stalls, loop unrolling to expose independent instructions, avoid instruction dependencies, avoid conditional branches etc.

New Instruction Set Architectures (ISA)

- **MMX** 2-wide float
- **SSE** 4-wide float, 2-wide double
- **AVX** 8-wide float, 4-wide double
- **New instructions** FMA (fused multiply accumulate)

Sequential Performance

Instruction Level Parallelism (ILP)

- **Out of order execution** re-order instructions
- **Pipelining** "assembly line" parallelism
- **Superscalar architecture** multiple execution units
- **Branch prediction** speculative execution

Compilers and processors already very good at this part!
How can we help? Keep data close to processor to avoid pipeline stalls, loop unrolling to expose independent instructions, avoid instruction dependencies, avoid conditional branches etc.

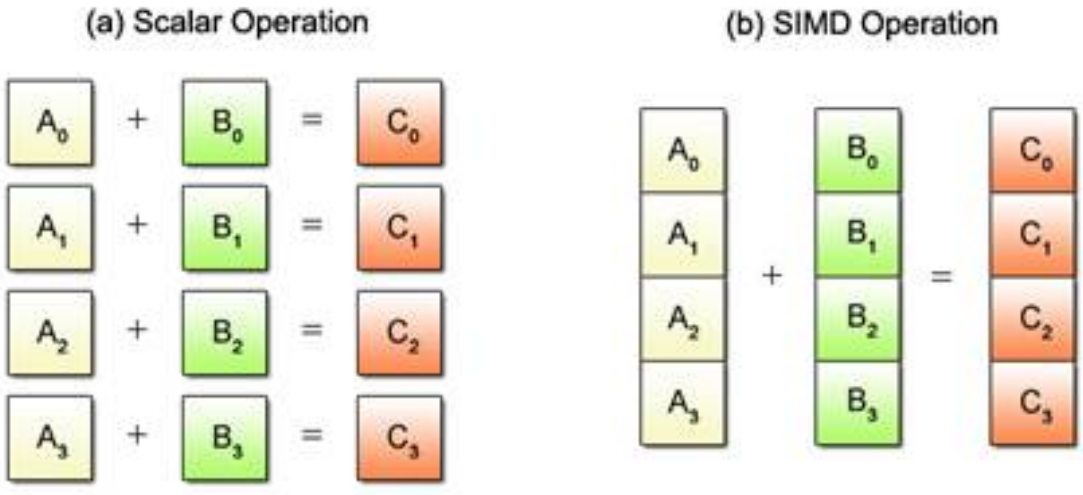
New Instruction Set Architectures (ISA)

- **MMX** 2-wide float
- **SSE** 4-wide float, 2-wide double
- **AVX** 8-wide float, 4-wide double
- **New instructions** FMA (fused multiply accumulate)

Many ways of doing this!

Vectorization

SIMD: Single Instruction Multiple Data



Steps

- Start thinking in vectors instead of scalars (float, double)
- Re-organize computations as vector operations
- Tell the compiler it is safe to use SIMD instructions

Implicit Vectorization

- **Auto-Vectorisation:** Loop unrolling, inlining, compiler flags (-O3, -march=native) and hints
  - Compiler specific extensions, not portable and no guarantee of vecorizing
  - #pragma ivdep (tell GCC to ignore vector dependency)
  - __builtin_assume_aligned(a, 32) (tell GCC array is aligned)
  - __assume_aligned(a, 32) (tell Intel compiler array is aligned)
  - -fopt-info-vec-optimized (vectorization report with GCC)
  - -qopt-report=2 (vectorization report with Intel)

Explicit Vectorization

- **OpenMP 4.0: SIMD pragma**
  - #pragma omp simd [clauses] (vectorize for loops)
  - clause: safelen(len) (vectors of length len are safe)
  - clause: aligned(v1,v2:alignment) (vectors are aligned)

Explicit Vectorization

- **OpenMP 4.0: SIMD pragma**
  - #pragma omp simd [clauses] (vectorize for loops)
  - clause: safelen(len) (vectors of length len are safe)
  - clause: aligned(v1,v2:alignment) (vectors are aligned)
- **Assembly:** too hard!

Explicit Vectorization

- **OpenMP 4.0: SIMD pragma**
  - #pragma omp simd [clauses] (vectorize for loops)
  - clause: safelen(len) (vectors of length len are safe)
  - clause: aligned(v1,v2:alignment) (vectors are aligned)
- **Assembly:** too hard!
- **Vector Intrinsics:** details on next slide

Explicit Vectorization

- **OpenMP 4.0: SIMD pragma**
  - #pragma omp simd [clauses] (vectorize for loops)
  - clause: safelen(len) (vectors of length len are safe)
  - clause: aligned(v1,v2:alignment) (vectors are aligned)
- **Assembly:** too hard!
- **Vector Intrinsics:** details on next slide
- **Vector Intrinsics (the C++ way)**
  - Vector objects, overloaded operatos (+, -, *, ||, && etc)
  - See file "intrin-wrapper.h" in https://github.com/NYU-HPC19/lecture3
  - Other implementations https://www.agner.org/optimize/#vectorclass
  - Similar proposals for future C++ standard library https://en.cppreference.com/w/cpp/experimental/simd

Vector Intrinsics

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

- **Aligned memory allocation:** reading from aligned arrays is faster, un-aligned reads require multiple instructions. Allocate arrays using,
  - void* aligned_malloc(int); (for dynamic memory)
  - alignas(64) char double[128]; (for static arrays)
- **Basic Vector Operations (AVX only)**
  - Vector type: __m256d (vector of 4-doubles)
  - Load aligned: _mm256_load_pd(double const *)
  - Load unaligned: _mm256_load_pd(double const *)
  - Store aligned: _mm256_store_pd(double const *, __m265d)
  - Store un-aligned: _mm256_store_pd(double const *, __m265d)
  - Vector Addition: _mm256_add_pd(__m265d, __m265d)
  - Vector Multiplication: _mm256_mul_pd(__m265d, __m265d)
  - Vector FMA: _mm256_fmadd_pd(__m265d, __m265d, __m265d)
  - Other intrinsics to know: permutation, comparison, bitwise operations, streaming reads/writes, prefetch instructions etc.

Special Functions

- Special functions like div, mod, $\exp$, $\sqrt{x}$, $\sin$, $\cos$ computed in software or special hardware units.
- Require $\mathcal{O}(10)$ or $\mathcal{O}(100)$ cycles to compute.
- Example: division requires 10 cycles and has latency of 38 cycles.
- Specialized algorithms for some computations (fast inverse sqrt used in Quake III)

Outline

Basic Linear Algebra Subprograms (BLAS)

Specification for basic linear algebra operations

- Level 1: vector-vector operations (dot-product, norm, axpy)
- Level 2: matrix-vector operations (generalized matrix-vector multiplication: gemv)
- Level 3: matrix-matrix operations (generalized matrix-matrix multiplication: gemm)

Many implementations available

- Open-source: GotoBLAS (Kazushige Goto), OpenBLAS, ATLAS (Automatically Tuned Linear Algebra Software)
- Intel Math Kernel Library (MKL)

GEMM (sgemm, dgemm)

Function declaration:

```
extern "C" { // from C++
void dgemm_(char* TRANSA, char* TRANSB,
    int* M, int* N, int* K, double* ALPHA,
    double* A, int* LDA, double* B, int*
    LDB, double* BETA, double* C, int* LDC);
}
```

Linking:

```
g++ MMult.cpp -lblas
icpc MMult.cpp -mkl
```

(for linking to MKL using non-Intel compilers: https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor)

Other Useful Libraries

Linear Algebra Package (LAPACK)

- Linear solvers, LU factorization, QR factorization, singular value decomposition (SVD)
- Built on top of BLAS
- Many implementations available (also included in MKL)

Fastest Fourier Transform in the West (FFTW)
http://www.fftw.org

- Optimized FFT implementation, open source.