How to Write Fast Code

18-645, spring 2008

5th Lecture, Jan. 30th

Instructor: Markus Püschel

TAs: Srinivas Chellappa (Vas) and Frédéric de Mesmay (Fred)

Technicalities

- Research project
- First homework:
  After your name, write number of hours you needed

Today

- Runtime/performance measurement of numerical code
- Cache behavior of code

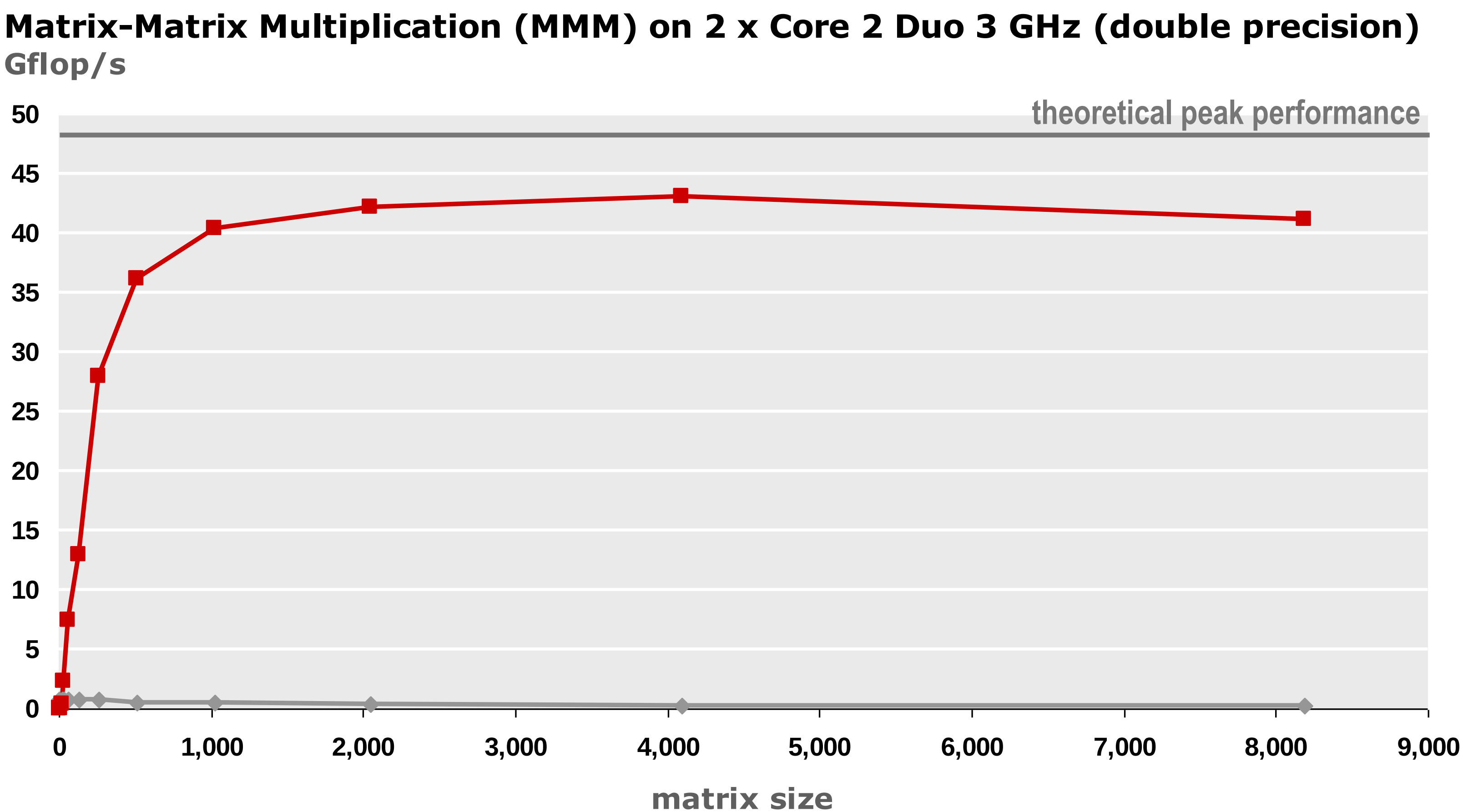
Runtime versus Performance

- **We consider numerical programs**
  - Example: Computing MMM by definition
  - Two measures: runtime and performance

- **Runtime**
  - Measured in seconds
  - Is what ultimately matters

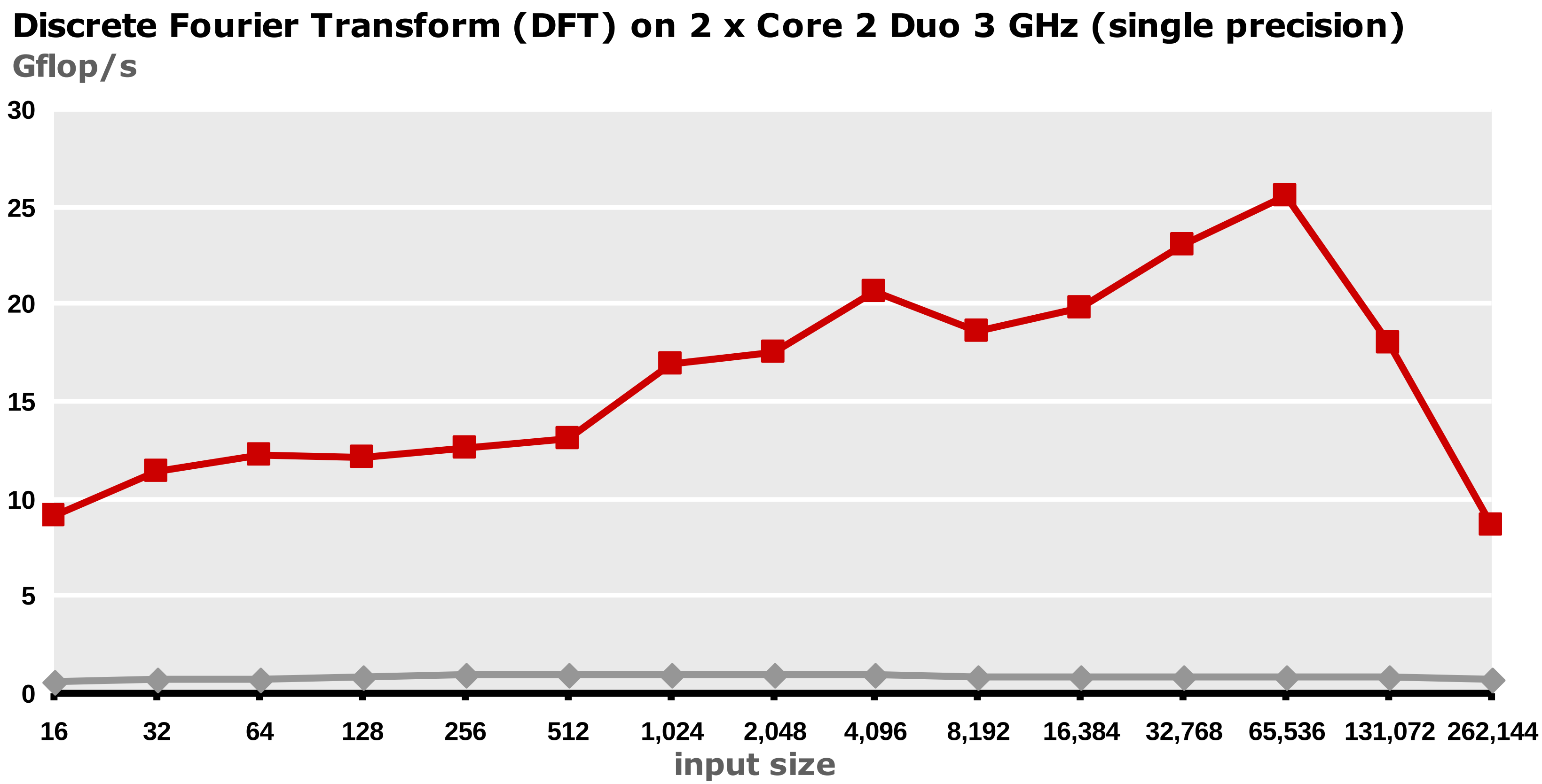
- **Performance**
  - Usually: measured in floating point operations per second = flop/s (or Mflop/s, Gflop/s)
  - Floating point operations = additions + multiplications (arithmetic cost)
  - Assumes negligible amount of divisions, sin, cos, ….
  - Gives you an idea how much room for improvement when comparing to theoretical peak performance of your machine
  - Careful: higher performance ≠ shorter runtime (Why?)

Example: MMM Performance



- Exact operations count is known: $2n^3$, so performance (here in Gflop/s) can be computed from runtime
- Fast code reaches 85% of peak!

Example: DFT Performance



- Exact operations count is not known: somewhere between 4 to $5n\log_2(n)$
- So $5n\log_2(n)$ is used in all cases: preserves runtime relationship
- Fast code reaches only up to 40 to 50% of peak, drop for large sizes

Summary

- **Showing performance is often preferrable to showing runtime**
  - If it is computed using the same flops (arithmetic cost) formula for all implementations
  - Preserves runtime relationship between different implementations (performance ≈ inverse runtime)
  - Gives an idea of absolute quality (how far from peak?)
  - Yields “higher is better” plots: psychologically preferrable to “lower is better” plots

- **Question: What percentage of peak is achievable for a given algorithm?**
- **Answer: It depends on**
  - Reuse (memory hierarchy)
  - Regular fine grain parallelism (vector instructions)
  - Coarse grain parallelism (multiple threads)

Reuse

- **Cache misses**
  - Deteriorate performance: Much more expensive than adds and mults
- **Ideally:**
  - Every data element is brought into cache once
  - All computation that needs it is performed before it is evicted from cache
  - Means only one compulsory miss
  - Miss time overcompensated by computation time, but there are limitations
- **Reuse: The reuse of an O(f(n)) algorithm is given by**

  O(f(n)/n)
  - Intuitively measures how often every input element is on average needed in the computation
  - Can also be measured exactly: Arithmetic cost of algorithm divided by n

CPU bound versus Memory bound

- **Definitions are not precise**

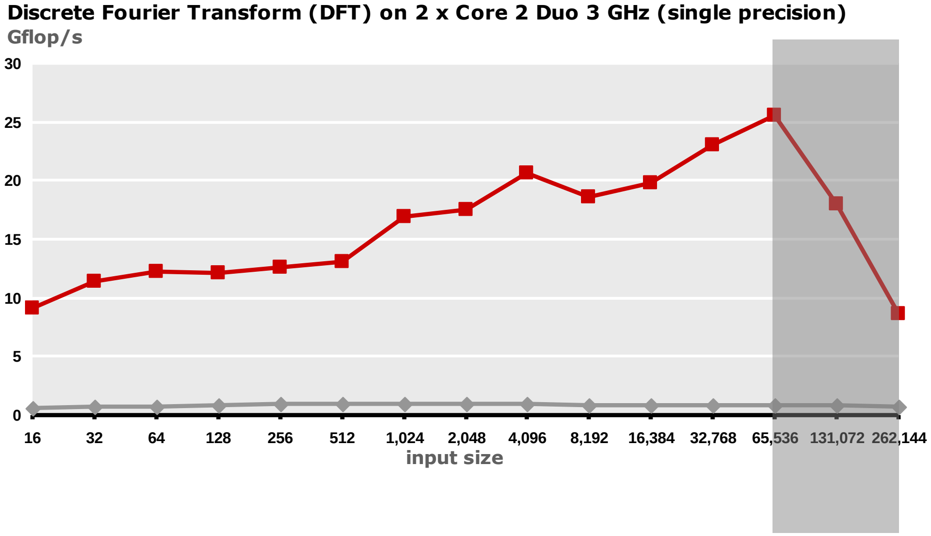
- **An algorithm with high reuse is called CPU bound**
  - Most time is spent computing
  - Will run faster if CPU is faster

- **An algorithm with low reuse is called memory bound**
  - Most time spent transferring data in the memory hierarchy
  - Will run faster if memory bus is faster

- **Examples: (blackboard)**
  - MMM, DFT, MVM

Effects

FFT: O(log(n)) reuse

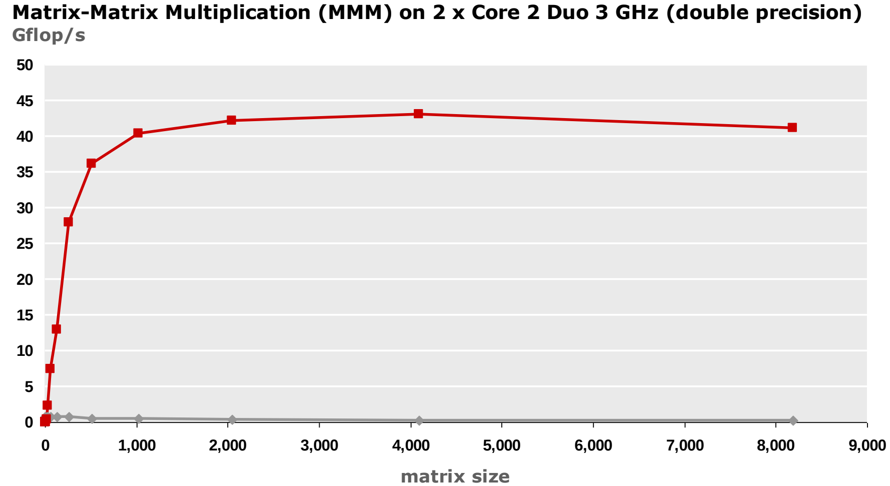


40-50% peak

Performance drop outside L2 cache

Most time spent transferring data

MMM: O(n) reuse



80-85% peak

Performance can be maintained

Cache miss time compensated/hidden by computation

Actual Benchmarking (Read Section 3.2 in Tutorial)

- **First: Verify your code!**

- **Measure runtime in seconds for a set of relevant input sizes**

- **Determine performance: flop/s**
  (number floating point ops/second)
  - Needs arithmetic cost:
    - Obtained statically (cost analysis since you understand the algorithm)
    - or dynamically (tool that counts, or replace ops by counters through macros)
  - Compare to theoretical peak performance
  - Careful: Different algorithms may have different op count, i.e., best flop/s is not always best runtime

Guide to benchmarking: How to measure runtime?

- **C clock()**
  - process specific, low resolution, very portable
- **gettimeofday**
  - measures wall clock time, higher resolution, somewhat portable
- **Performance counter (e.g., TSC on Pentiums)**
  - measures cycles (i.e., also wall clock time), highest resolution, not portable

- **Careful:**
  - measure only what you want to measure
  - ensure proper machine state
    (e.g., cold or warm cache = input data is or is not in cache)
  - measure enough repetitions
  - check how reproducible; if not reproducible: fix it
- **Getting proper measurements is not easy at all!**

Example: Timing MMM

- Assume `MMM(A,B,C,n)` computes
  `C = C + AB`, `A,B,C` are `nxn` matrices

```
double time_MMM(int n)
{ // allocate
  double *A=(double*)malloc(n*n*sizeof(double));
  double *B=(double*)malloc(n*n*sizeof(double));
  double *C=(double*)malloc(n*n*sizeof(double));

  // initialize
  for(int i=0; i<n*n; i++){
    A[i] = B[i] = C[i] = 0.0;
  }

  init_MMM(A,B,C,n); // if needed

  // warm up cache (for warm cache timing)
  MMM(A,B,C,n);

  // time
  ReadTime(t0);
  for(int i=0; i<TIMING_REPETITIONS; i++)
    MMM(A,B,C,n);
  ReadTime(t1);

  // compute runtime
  return (double)((t1-t0)/TIMING_REPETITIONS);
}
```

Problems with Timing

- Too few iterations: inaccurate non-reproducible timing
- Too many iterations: system events interfere
- Machine is under load: produces side effects
- Multiple timings performed on the same machine
- Bad data alignment of input/output vectors: align to multiples of cache line (on Core: address is divisible by 64)
- Time stamp counter (if used) overflows
- Machine was not rebooted for a long time: state of operating system causes problems
- Computation is input data dependent: choose representative input data
- Computation is inplace and data grows until an exception is triggered (computation is done with NaNs)
- You work on a laptop that has dynamic frequency scaling
- Solution: check whether timings make sense, are reproducible

Cache Behavior of Code

- **Blackboard**
  - Small example
  - Data reuse and neighbor reuse
  - Sequential access
  - Strided access