Computer Algorithms and Architectures

William D. Gropp
Mathematics and Computer Science
www.mcs.anl.gov/~gropp

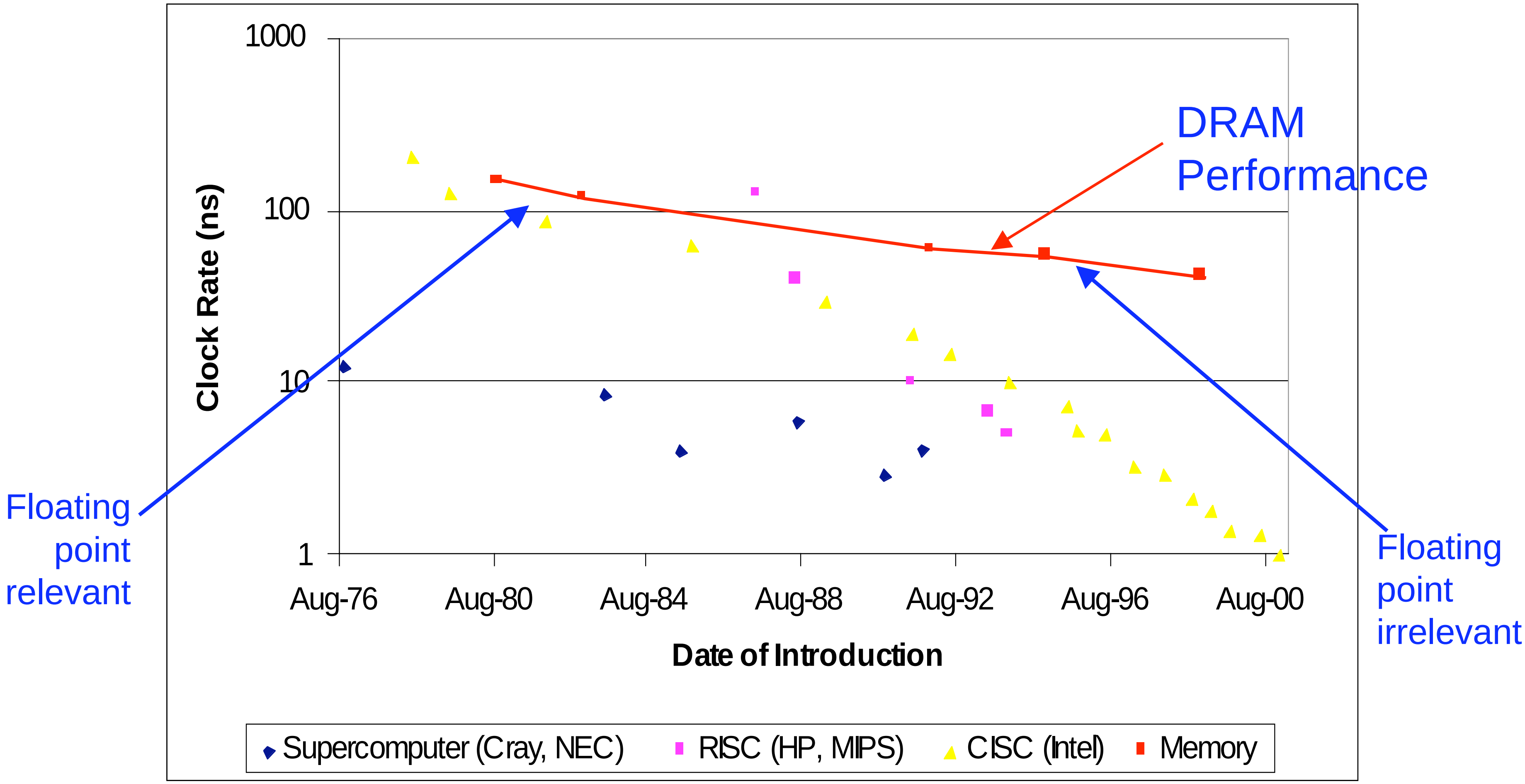
Algorithms

- What is an algorithm?
  - A set of instructions to perform a task
- How do we evaluate an algorithm?
  - Correctness
  - Accuracy
    - Not an absolute
  - Efficiency
    - Relative to current and future machines
- How do we measure efficiency?
  - Often by counting floating point operations
  - Compare to “peak performance”

Real and Idealized Computer Architectures

- Any algorithm assumes an idealized architecture
  - ♦ Common choice:
    - Floating point work costs time
    - Data movement is free
  - ♦ Real systems:
    - Floating point is free (fully overlapped with other operations)
    - Data movement costs time…a *lot* of time
- Classical complexity analysis for numerical algorithms is *no longer correct* (more precisely, no longer *relevant*)
  - ♦ Known since at least BLAS2 and BLAS3

CPU and Memory Performance



Trends in Computer Architecture I

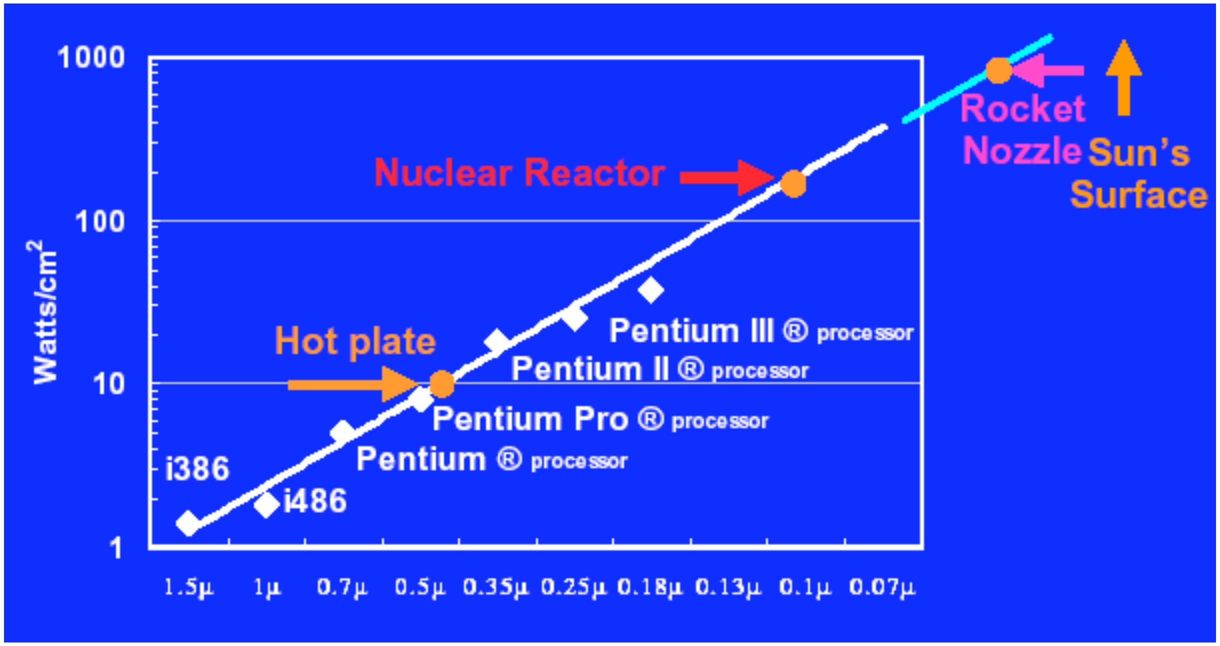
- Latency to memory will continue to grow relative to CPU speed
  - ♦ Latency hiding techniques require finding increasing amounts of independent work: Little's law implies
    - Number of concurrent memory references = Latency * rate
    - For 1 reference per cycle, this is already 100–1000 concurrent references

Trends in Computer Architecture II

- Clock speeds will continue to increase
  - ♦ The rate of clock rate increase has increased recently ☺
  - ♦ Light travels 3 cm (in a vacuum) in one cycle of a 10 GHz clock
    - CPU chips won't be causally connected within a single clock cycle, i.e., a signal will not cross the chip in a single clock cycle
    - Processors will be parallel!

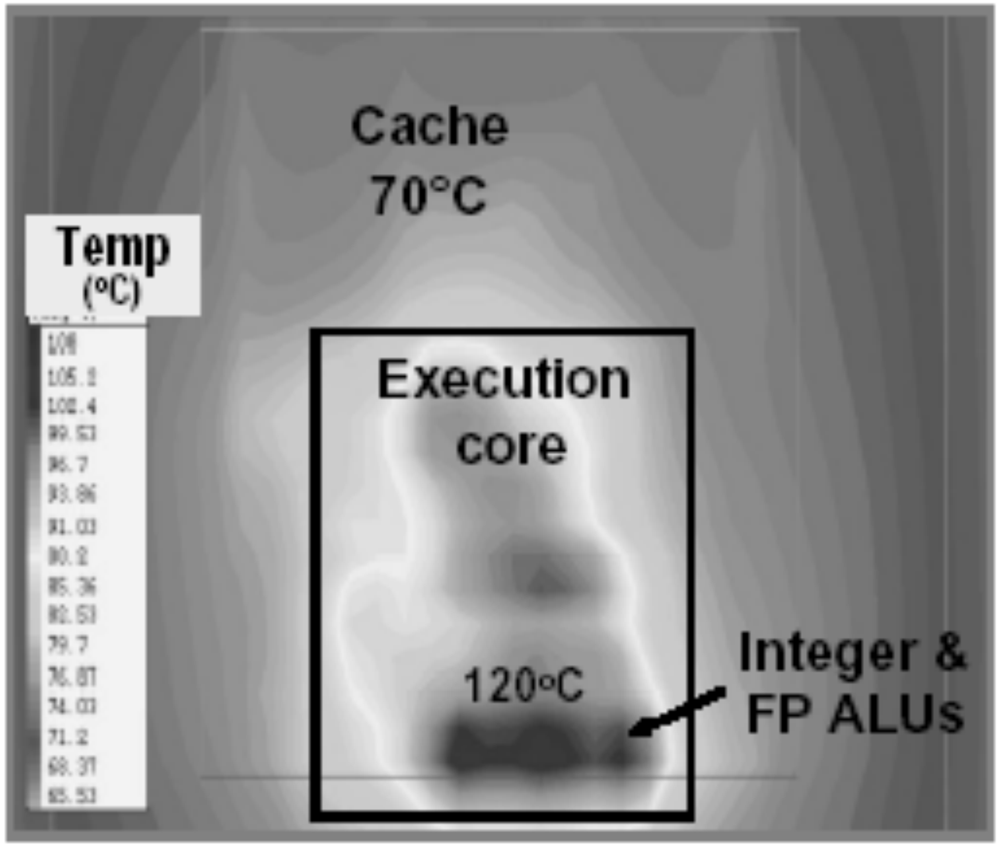
Trends in Computer Architecture III

- Power dissipation problems will force more changes
  - ♦ Current trends imply chips with energy densities greater than a nuclear reactor
  - ♦ Already a problem: In 2003, an issue of consumer reports looks at the likelihood of getting a serious burn from your laptop!
  - ♦ Will force new ways to get performance, such as extensive parallelism



Itanium Power Dissipation

- Power is not uniformly distributed across chip
- Peak power densities growing even faster



Consequences

- Gap between memory and processor performance will continue to grow
- Data motion will dominate the cost of many (most) calculations
- The key is to find a computational cost abstraction that is as simple as possible *but no simpler*

Architecture Invariants

- Performance is determined by memory performance
- Memory system design for performance makes system performance less predictable
- Fast memories possible, but
  - Expensive ($)
  - Large (meters^3)
  - Power hungry (Watts)
- Algorithms that don't take these realities into account may be irrelevant

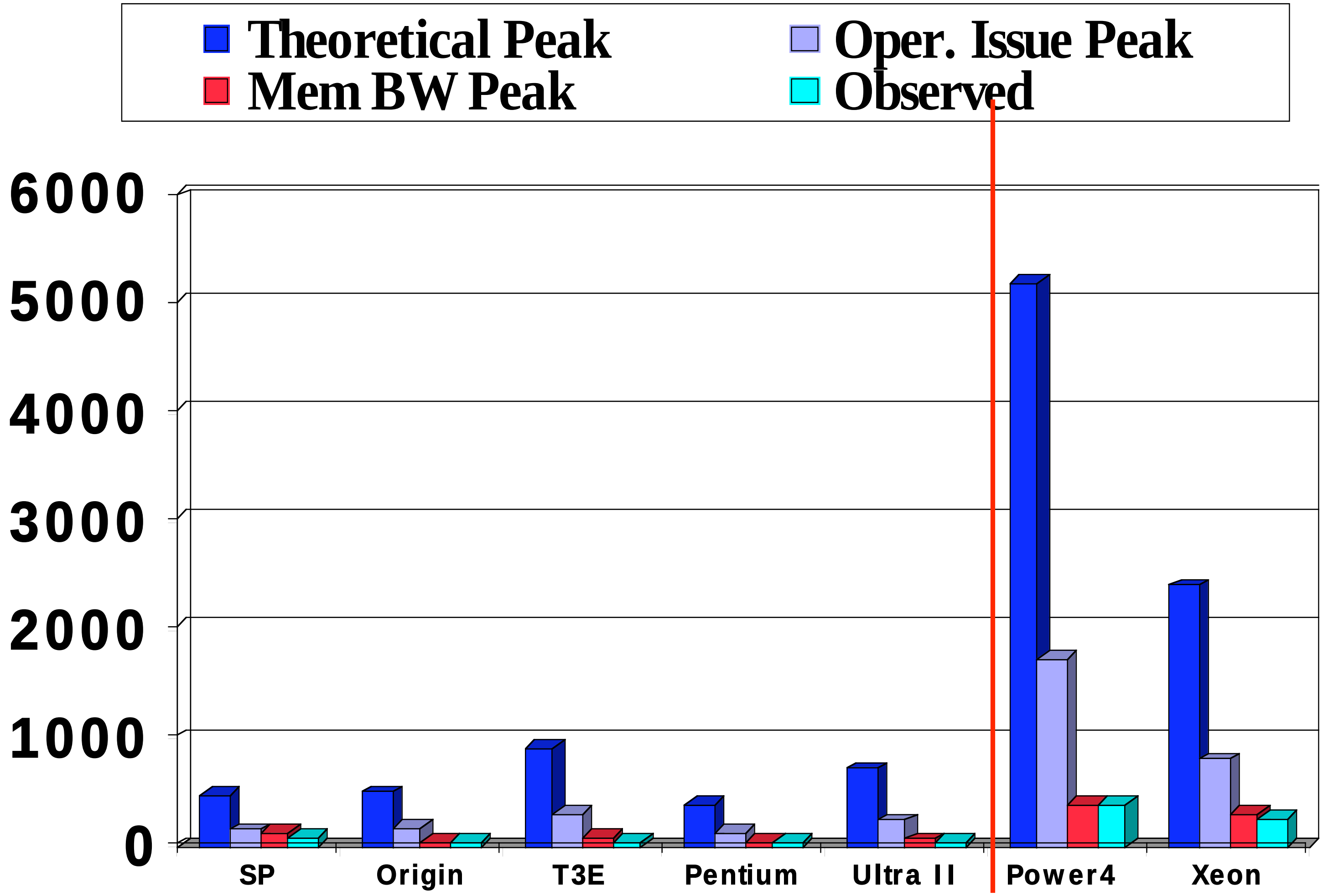
Node Performance

- Current laptops now have a peak speed (based on clock rate) of over 2 Gflops (20 Cray1s!)
- Observed (sustained) performance is often a small fraction of peak
- Why is the gap between "peak" and "sustained" performance so large?
- Lets look at a simple numerical kernel- sparse matrix-vector multiply

Realistic Measures of Peak Performance

Sparse Matrix Vector Product

one vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



What About CPU-Bound Operations?

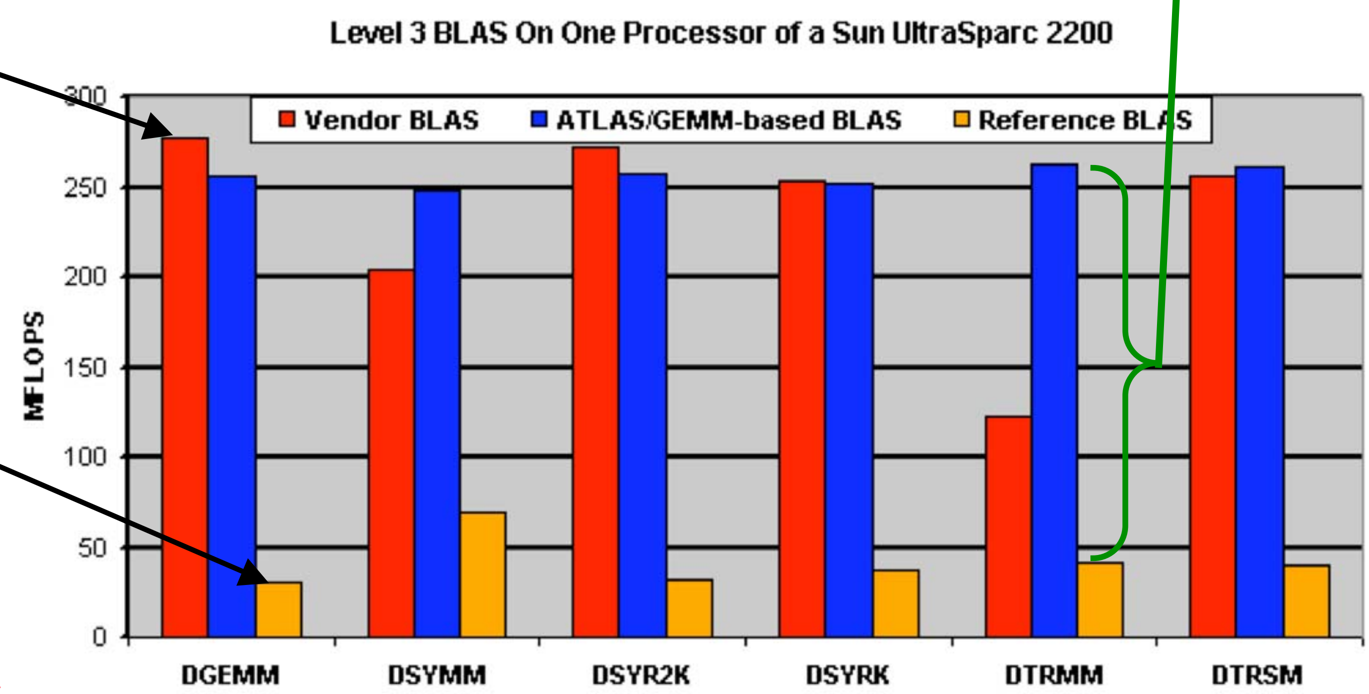
- Dense Matrix-Matrix Product
  - ♦ Most studied numerical program by compiler writers
  - ♦ Core of some important applications
  - ♦ More importantly, the core operation in High Performance Linpack
    - Benchmark used to “rate” the top 500 fastest systems
  - ♦ Should give optimal performance…

The Compiler Will Handle It (?)

Large gap between natural code and specialized code

Hand-tuned

Compiler



From Atlas

Enormous effort required to get good performance

Performance for Real Applications

- Dense matrix-matrix example shows that even for well-studied, compute-bound kernels, compiler-generated code achieves only a small fraction of available performance
  - ♦ “Fortran” code uses “natural” loops, i.e., what a user would write for most code
  - ♦ Others use multi-level blocking, careful instruction scheduling etc.
- Algorithms design also needs to take into account the capabilities of the *system*, not just the processor
  - ♦ Example: Cache-Oblivious Algorithms (http://supertech.lcs.mit.edu/cilk/papers/abstracts/abstract4.html)

The Computer As Labor-Saving Device

- Most current approaches to developing high-performance software are based on either
  - ♦ Compiler performs miracle
  - ♦ "Heroic" (and burned out) programmer
- Many of these techniques use transformations that can be mechanically applied, but require some programmer guidance.
  - ♦ Use the computer to apply these!
    - (Why is this so surprising?)
  - ♦ Examples include ATLAS (dense linear algebra), FFTW, PhiPac
  - ♦ New projects include SALSA (Self-Adaptive Linear Solver Architecture)
    - Joint work with Eijkhout, Dongarra, Keyes
    - Includes guides for choosing preconditioners, orderings, decomposition

Conclusions

- Performance models should count data motion, not flops
- Computers will continue to have multiple levels of memory hierarchy
  - Algorithms should *exploit* them
- Computers will be parallel
  - Algorithms can make effective use of greater adaptivity to give better time-to-solution and accuracy
- Denial is not a solution