Advanced Topics in Numerical Analysis: High Performance Computing

Sequential performance

Georg Stadler Dhairya Malhotra
Courant Institute, NYU
stadler@cims.nyu.edu dm4340@nyu.edu

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

Feb 11, 2019

Outline

Organization issues

- No class next week due to president's day.
- Differently from what was announced last week, there will be a class on Feb. 25. Our colleague Marsha Berger will (thankfully) cover for us and introduce the distributed memory model (OpenMP).
- There will be a new homework assignment (keep an eye on the Slack #homework channel). Will be due on March 4.
- There will be a class similar to this (taught by Ben Peherstorfer (Courant-CS)) a year from now.

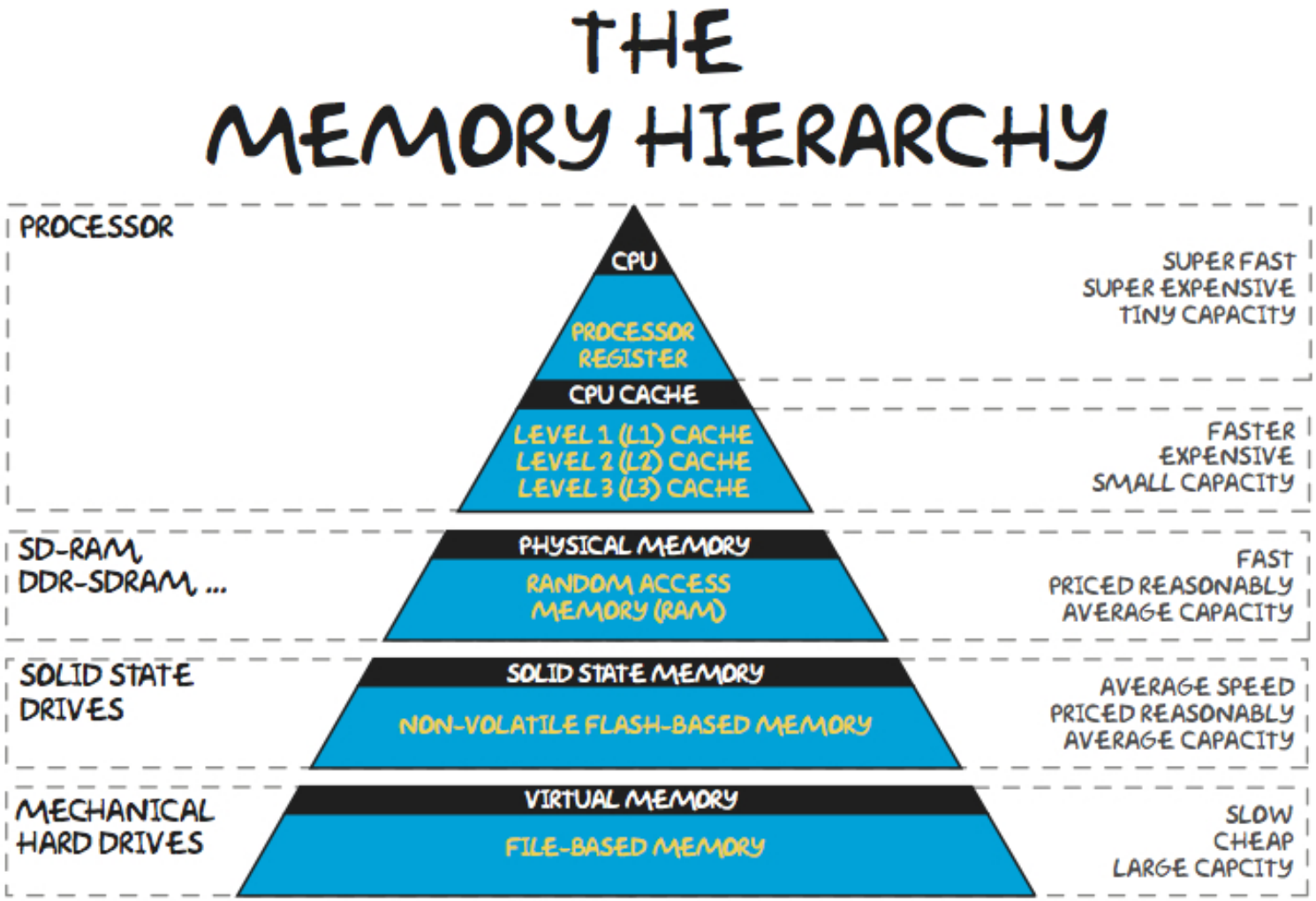
Topics today

- Valgrind and cachegrid demonstration (see also the video!)
- Single core optimization: pipelining, vectorization
- Tool of the week: Git
- Parallel machines and programming models

Outline

Memory hierarchies

On my Mac Book Pro: 32KB L1 Cache, 256KB L2 Cache, 3MB Cache, 8GB RAM



CPU: $\mathcal{O}(1\text{ns})$, L2/L3: $\mathcal{O}(10\text{ns})$, RAM: $\mathcal{O}(100\text{ns})$, disc: $\mathcal{O}(10\text{ms})$

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);
- cache-hit: required data is available in cache $\Rightarrow$ fast access

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);
- cache-hit: required data is available in cache $\Rightarrow$ fast access
- cache-miss: required data is not in cache and must be loaded from main memory (RAM) $\Rightarrow$ slow access

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);
- cache-hit: required data is available in cache $\Rightarrow$ fast access
- cache-miss: required data is not in cache and must be loaded from main memory (RAM) $\Rightarrow$ slow access

Computer architecture is complicated. We need a basic performance model.

- Processor needs to be "fed" with data to work on.
- Memory access is slow; memory hierarchies help.

Memory hierarchy

Simple model

1. Only consider two levels in hierarchy, fast (cache) and slow (RAM) memory
2. All data is initially in slow memory
3. Simplifications:
   - Ignore that memory access and arithmetic operations can happen at the same time
   - assume time for access to fast memory is 0
4. Computational intensity: flops per slow memory access

$$q = \frac{f}{m}, \text{ where } f \ldots \#\text{flops}, m \ldots \#\text{slow memop}.$$

Computational intensity should be as large as possible.

Memory hierarchy

Example: Matrix-matrix multiply Comparison between naive and blocked optimized matrix-matrix multiplication for different matrix sizes: Different algorithms can increase the computational intensity significantly.
BLAS: Optimized Basic Linear Algebra Subprograms

Memory hierarchy

Example: Matrix-matrix multiply Comparison between naive and blocked optimized matrix-matrix multiplication for different matrix sizes: Different algorithms can increase the computational intensity significantly.
BLAS: Optimized Basic Linear Algebra Subprograms

- Temporal and spatial locality is key for fast performance.
- Since arithmetic is cheap compared to memory access, one can consider making extra flops if it reduces the memory access.
- In distributed-memory parallel computations, the memory hierarchy is extended to data stored on other processors, which is only available through communication over the network.

Valgrind and cachegrind

Valgrind

- memory management tool and suite for debugging, also in parallel
- profiles heap (not stack) memory access
- simulates a CPU in software
- running code with valgrind makes it slower by factor of 10-100
- Documentation: http://valgrind.org/docs/manual/
- Examples: https://github.com/NYU-HPC19/lecture2

memcheck	cachegrind	callgrind
finds leaks	cache profiler	extension to
inval. mem. access	sources of cache	cachegrind
uninitialize mem.	misses	function call graph
incorrect mem. frees		

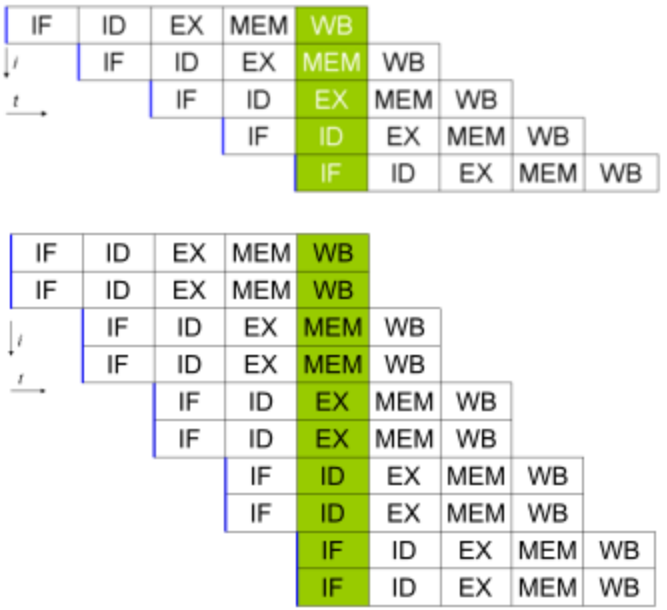
Outline

Levels of parallelism

- Parallelism at the bit level (64-bit operations)

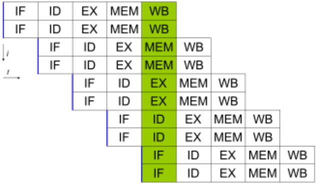
Levels of parallelism

- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle



Levels of parallelism

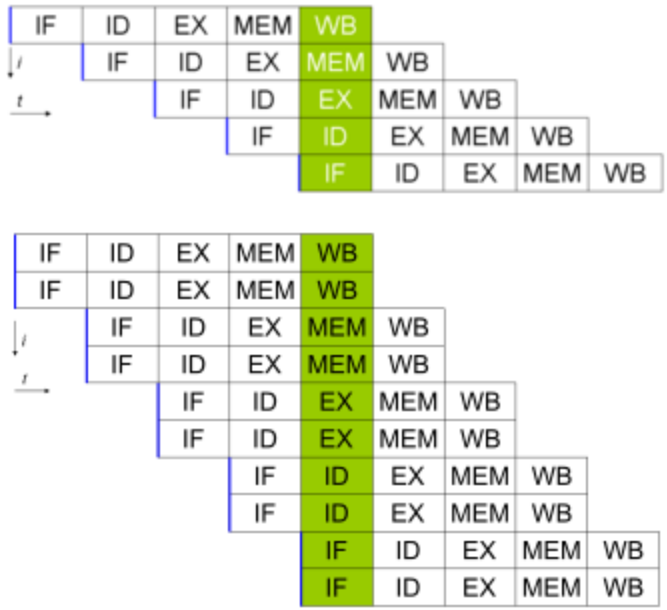
- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,...



all of the above assume single sequential control flow

Levels of parallelism

- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- multiple functional units parallelism: ALUs (algorithmic logical units), FPUs (floating point units), load/store memory units,...



all of the above assume single sequential control flow

- process/thread level parallelism: independent processor cores, multicore processors; parallel control flow

Work Depth Model

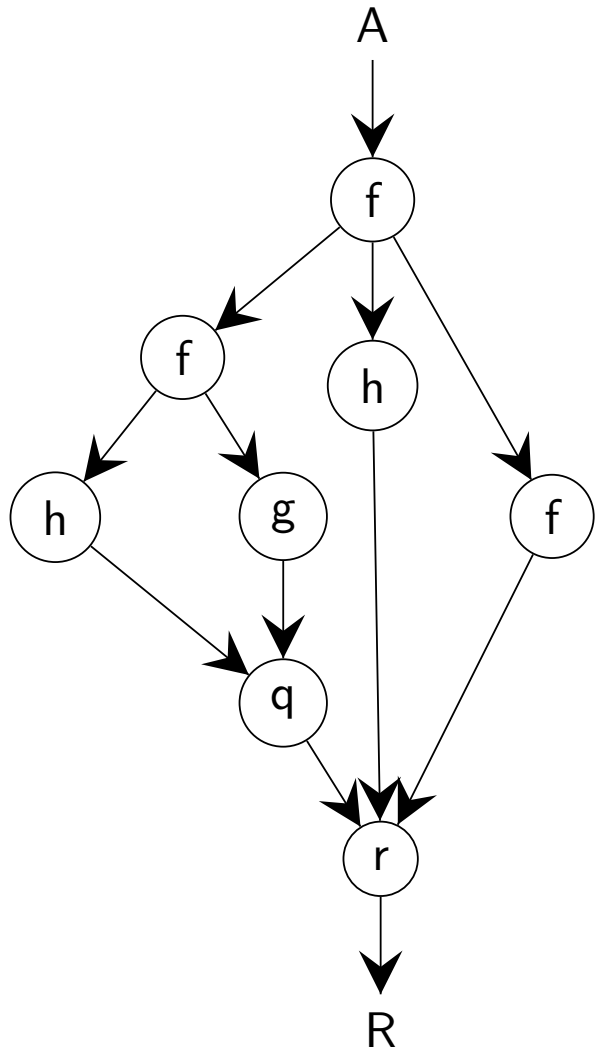
Is this parallel?

```
B=f(A);
C=f(B);
D=h(B);
G=h(C);
F=g(C);
E=f(B);
H=q(G, F);
R=r(H,D,E);
```

Work Depth Model

Is this parallel?

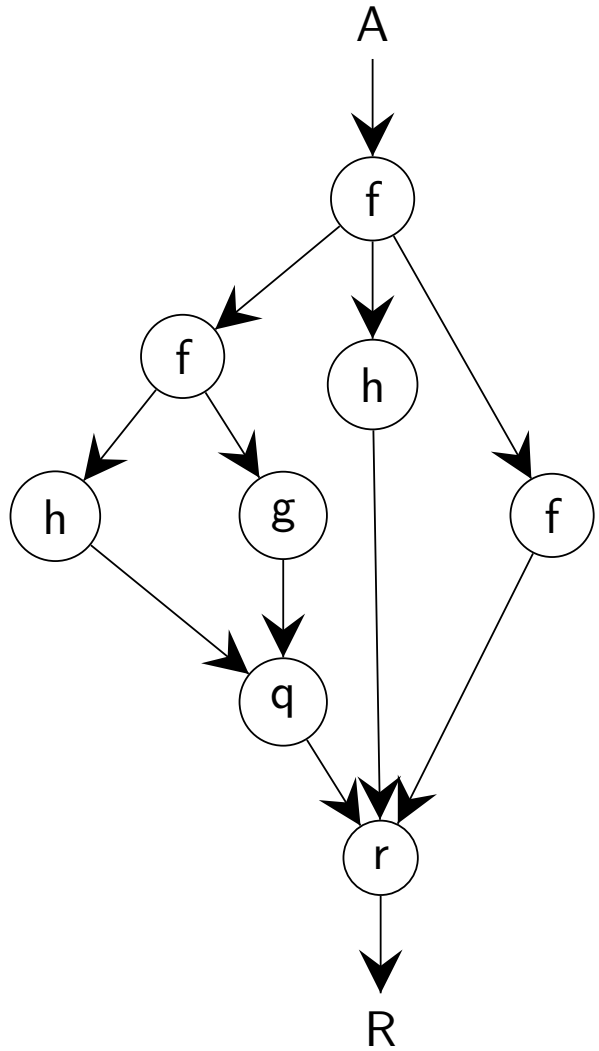
B=f(A);
C=f(B); D=h(B);
G=h(C); F=g(C); E=f(B);
H=q(G, F);
R=r(H,D,E);
How about now?



Work Depth Model

Is this parallel?

B=f(A);
C=f(B); D=h(B);
G=h(C); F=g(C); E=f(B);
H=q(G, F);
R=r(H,D,E);
How about now?



Directed Acyclic Graph (DAG)

Work = #-of-nodes
Depth = #-of-levels
Parallelism = Work/Depth

Work Depth Model

Reduction: $y = x_0 + x_1 + x_2 + \cdots + x_n$

work $= \mathcal{O}(n)$

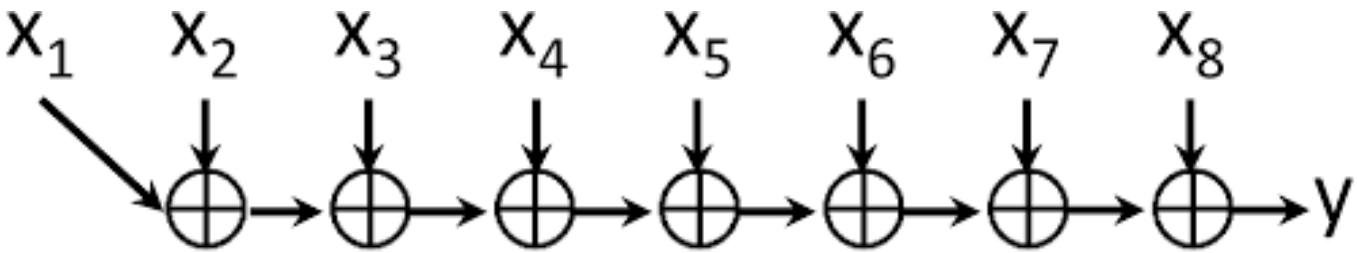
depth $= \mathcal{O}(n)$ or $\mathcal{O}(\log n)$

Work Depth Model

Reduction: $y = x_0 + x_1 + x_2 + \cdots + x_n$

work $= \mathcal{O}(n)$

depth $= \mathcal{O}(n)$ or $\mathcal{O}(\log n)$

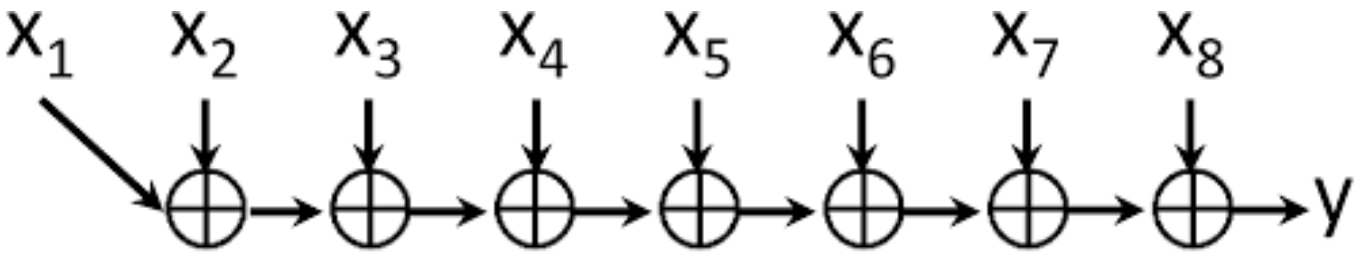


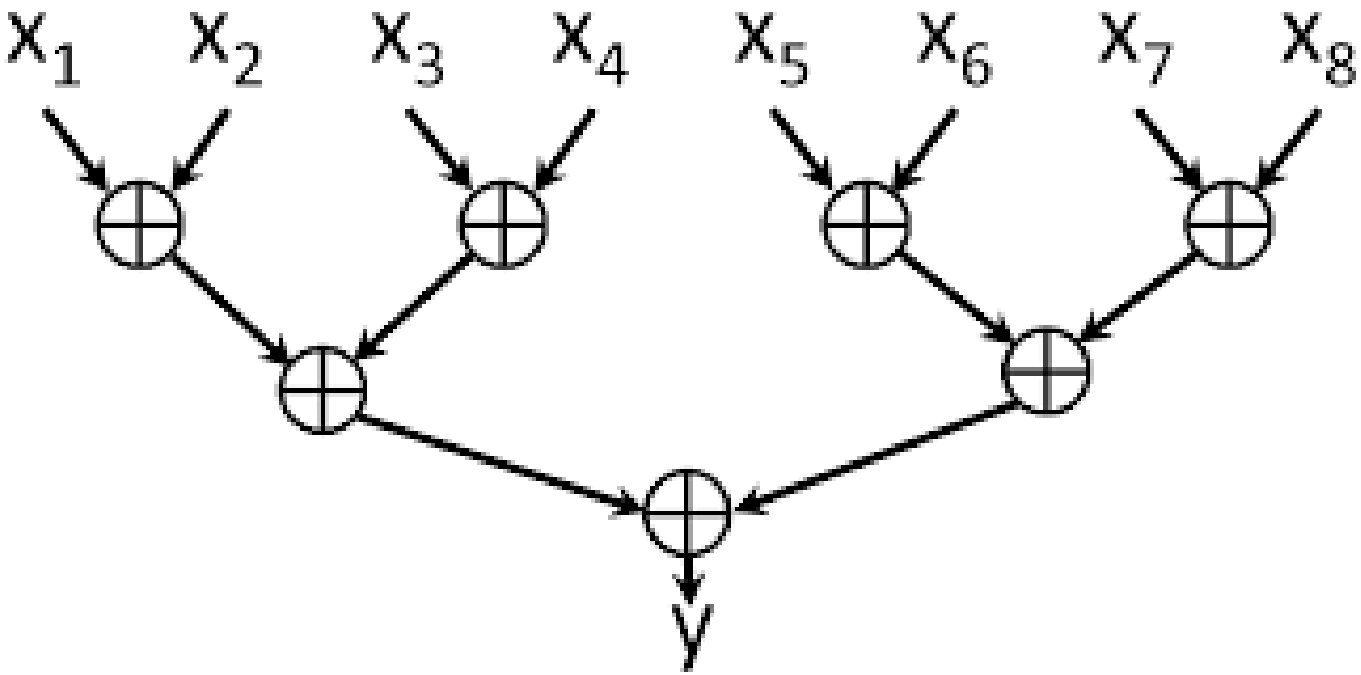
Work Depth Model

Reduction: $y = x_0 + x_1 + x_2 + \cdots + x_n$

work $= \mathcal{O}(n)$

depth $= \mathcal{O}(n)$ or $\mathcal{O}(\log n)$





Work Depth Model

Reduction: $y = x_0 + x_1 + x_2 + \cdots + x_n$

work $= \mathcal{O}(n)$

depth $= \mathcal{O}(n)$ or $\mathcal{O}(\log n)$

Matrix-Matrix Multiplication:

work $= \mathcal{O}(n^3)$

depth $= \mathcal{O}(\log n)$

parallelism $= \mathcal{O}(n^3/\log n)$

Work Depth Model

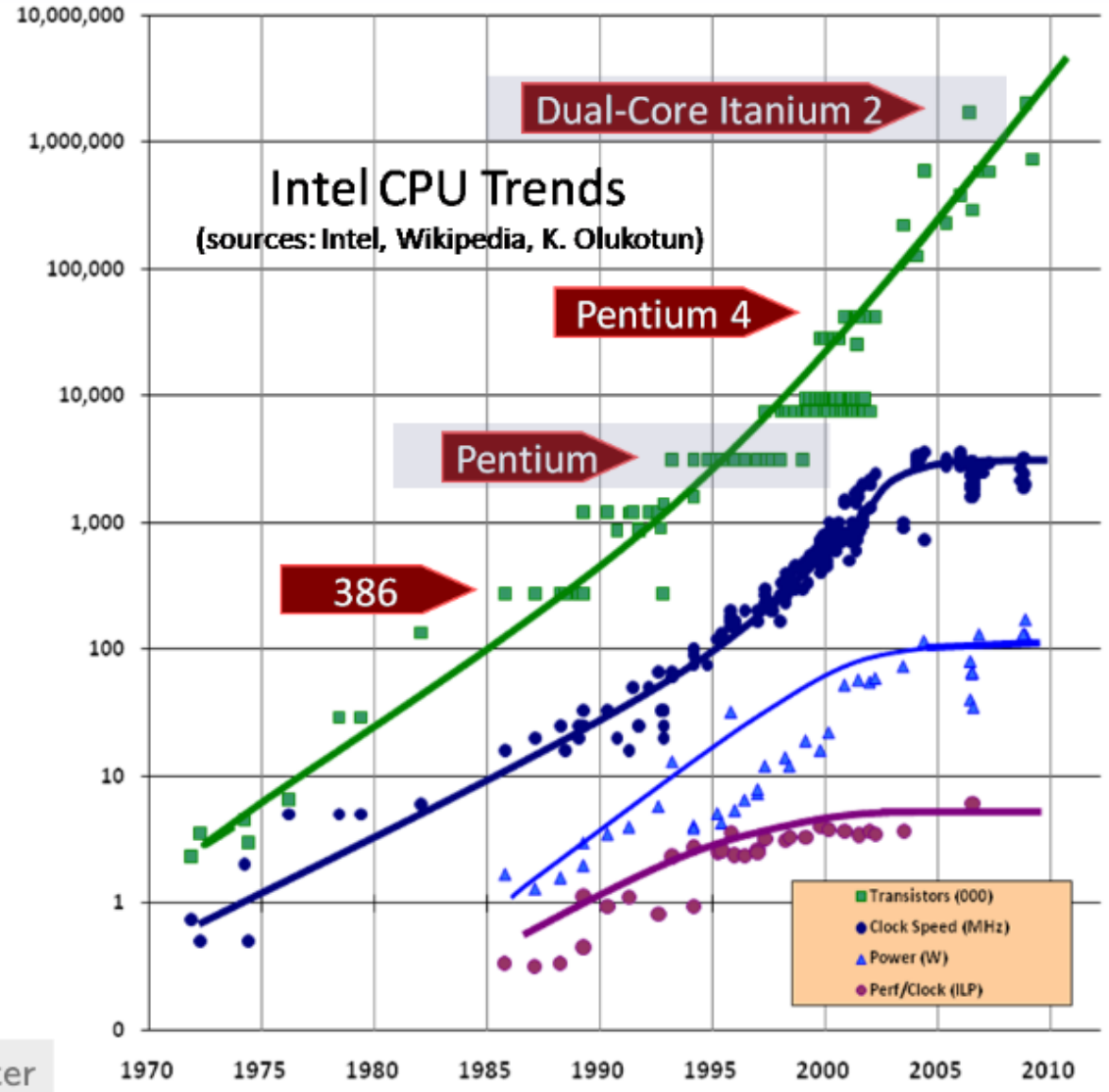
Reduction: $y = x_0 + x_1 + x_2 + \cdots + x_n$
work $= \mathcal{O}(n)$
depth $= \mathcal{O}(n)$ or $\mathcal{O}(\log n)$

Matrix-Matrix Multiplication:
work $= \mathcal{O}(n^3)$
depth $= \mathcal{O}(\log n)$
parallelism $= \mathcal{O}(n^3/\log n)$

Sorting:
work $= \mathcal{O}(n \log n)$
depth $= \mathcal{O}(\log^2 n)$
parallelism $= \mathcal{O}(n/\log n)$

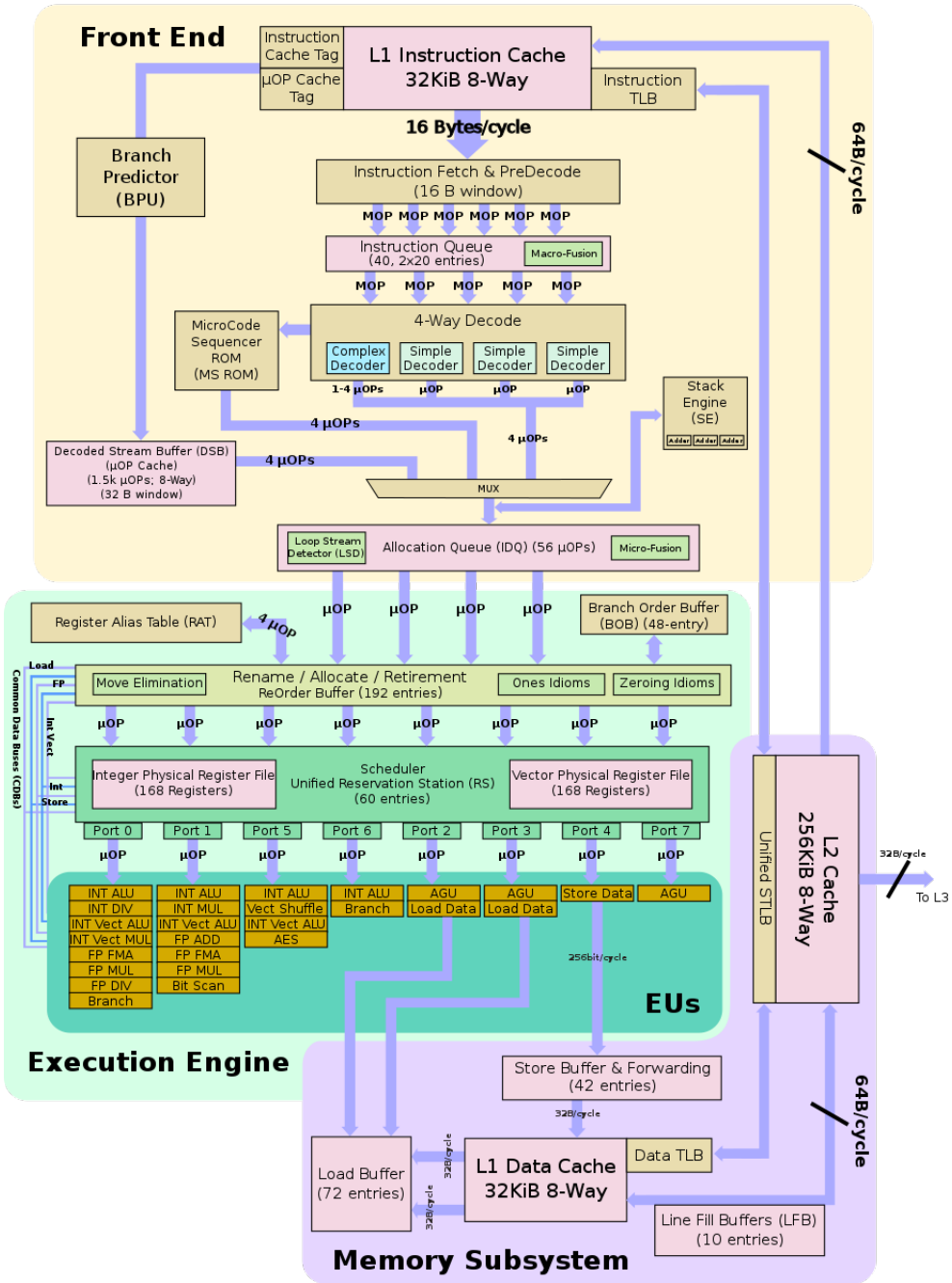
Sequential Performance

Exponential growth in number of transistors. What do these extra transistors do?

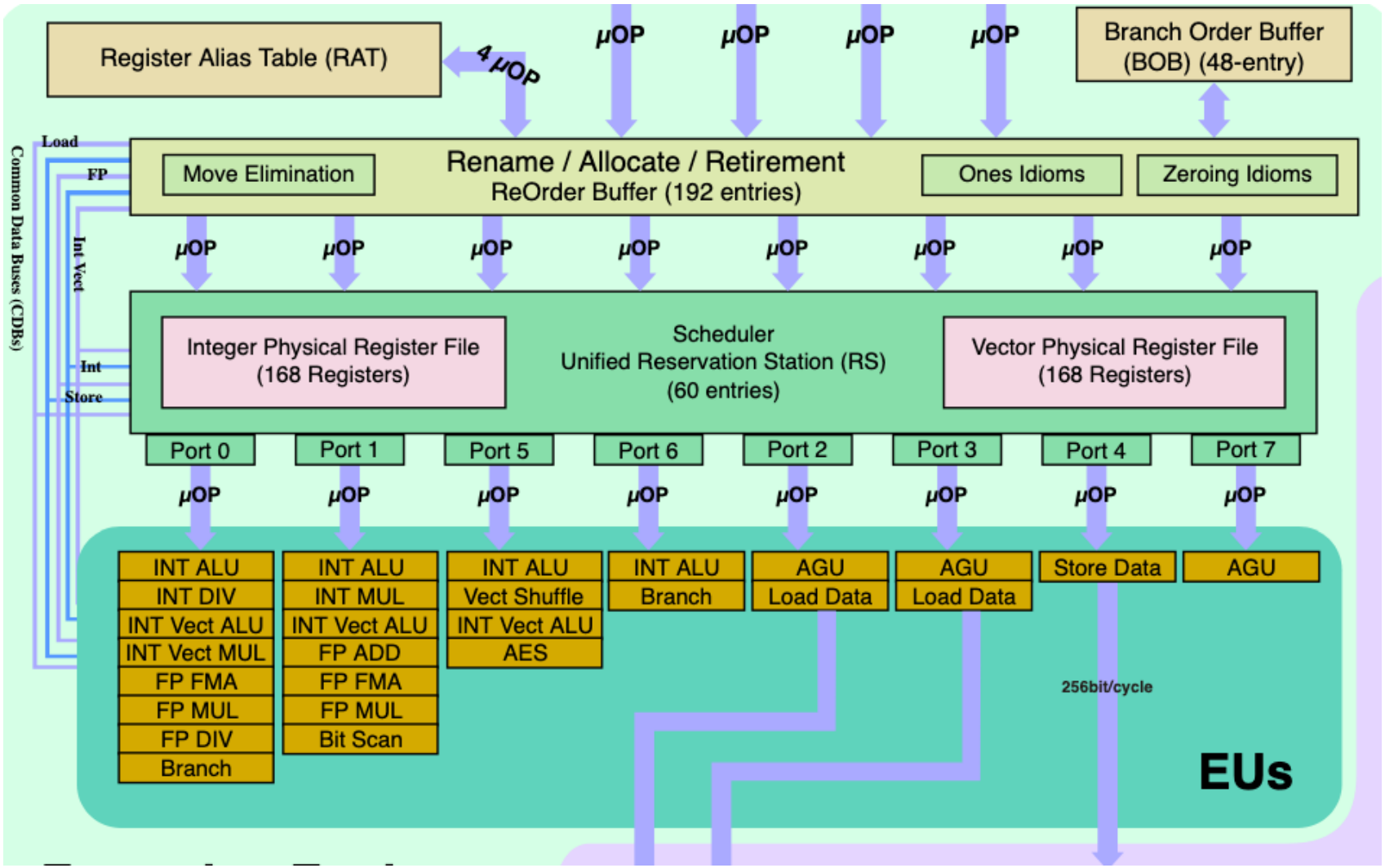


Herb Sutter

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.

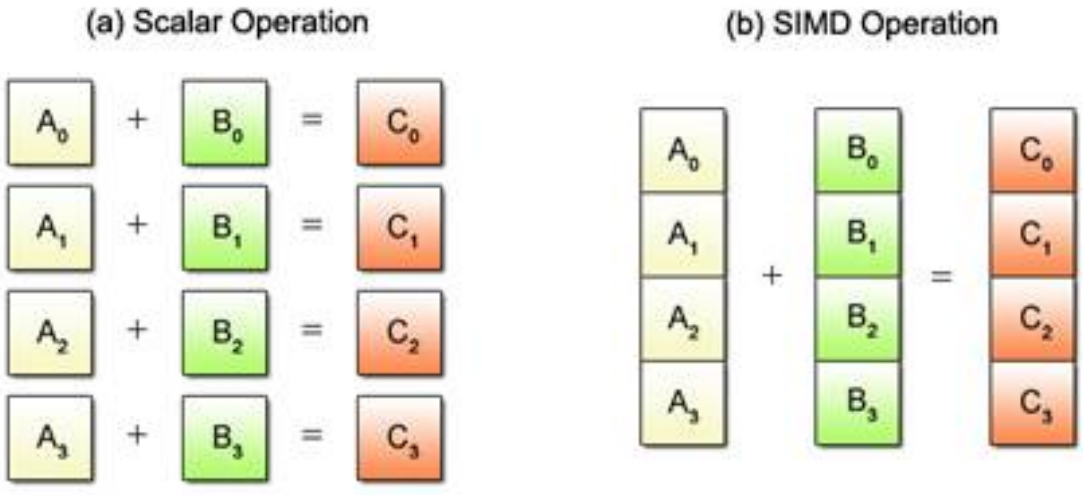
New Instruction Set Architectures (ISA)

- **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 Intruction 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)