The High Hanging Fruit

Dean Tullsen
UCSD

The Dilemma

- The parallelism crisis has the feel of a relatively *new* problem
  - Results from a huge technology shift
  - Has suddenly become pervasive
  - Carries extreme urgency – our ability to continue to scale performance is now completely tied to our ability to find parallelism.
  - Many researchers rushing in to work on the problem.
- But it is a very *old* problem
  - Smart people have been thinking about and building parallel machines for about 6 decades.

As a result

- We are faced with a “new”, critically urgent problem, but with all of the low hanging fruit stripped clean.
- Few easy solutions remain.

Some deep reservoirs of untapped parallelism

- Parallel speedup of sequential code
- Small pockets of parallelism
- Unpowered transistors

Sequential Code Will Always Be Critically Important

- Many important algorithms inherently sequential.

- Amdahl's Law tells us that eventually, the sequential code always dominates.

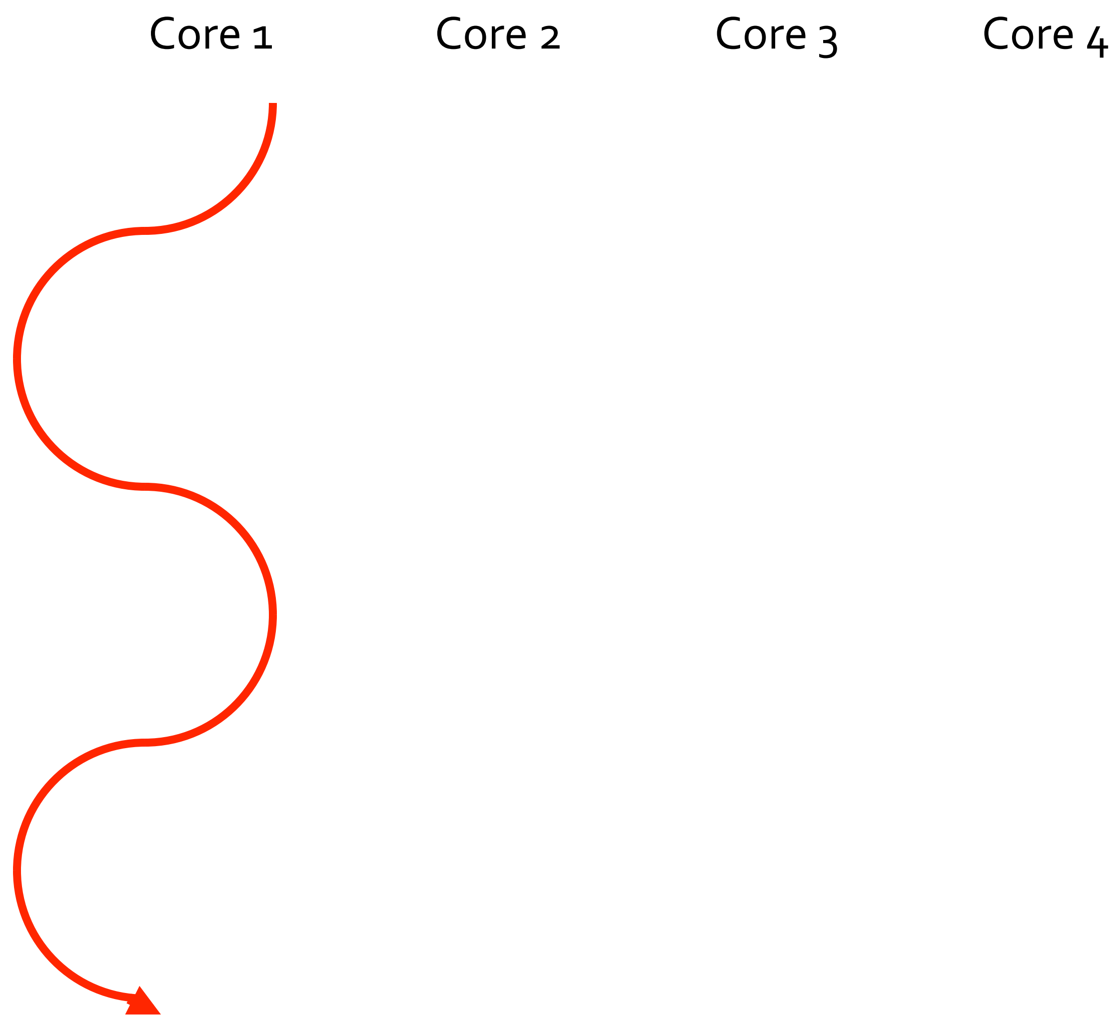
Parallel Speedup of Sequential Code

- We've been referring to this as "non-traditional parallelism"

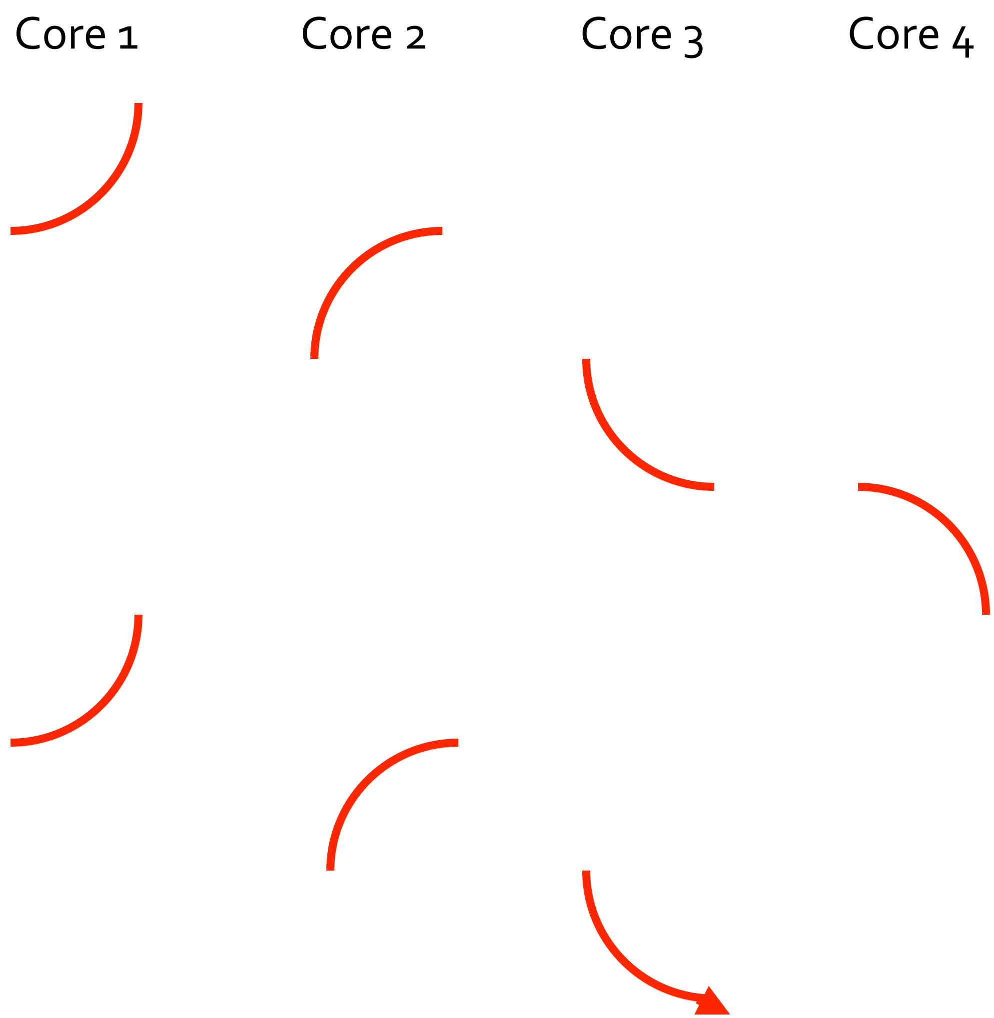
- Simply stated – how do you use multiple hardware contexts to run sequential code faster than a single context?

- Can we run sequential code faster on a machine optimized for parallel execution than on a machine optimized for sequential execution?

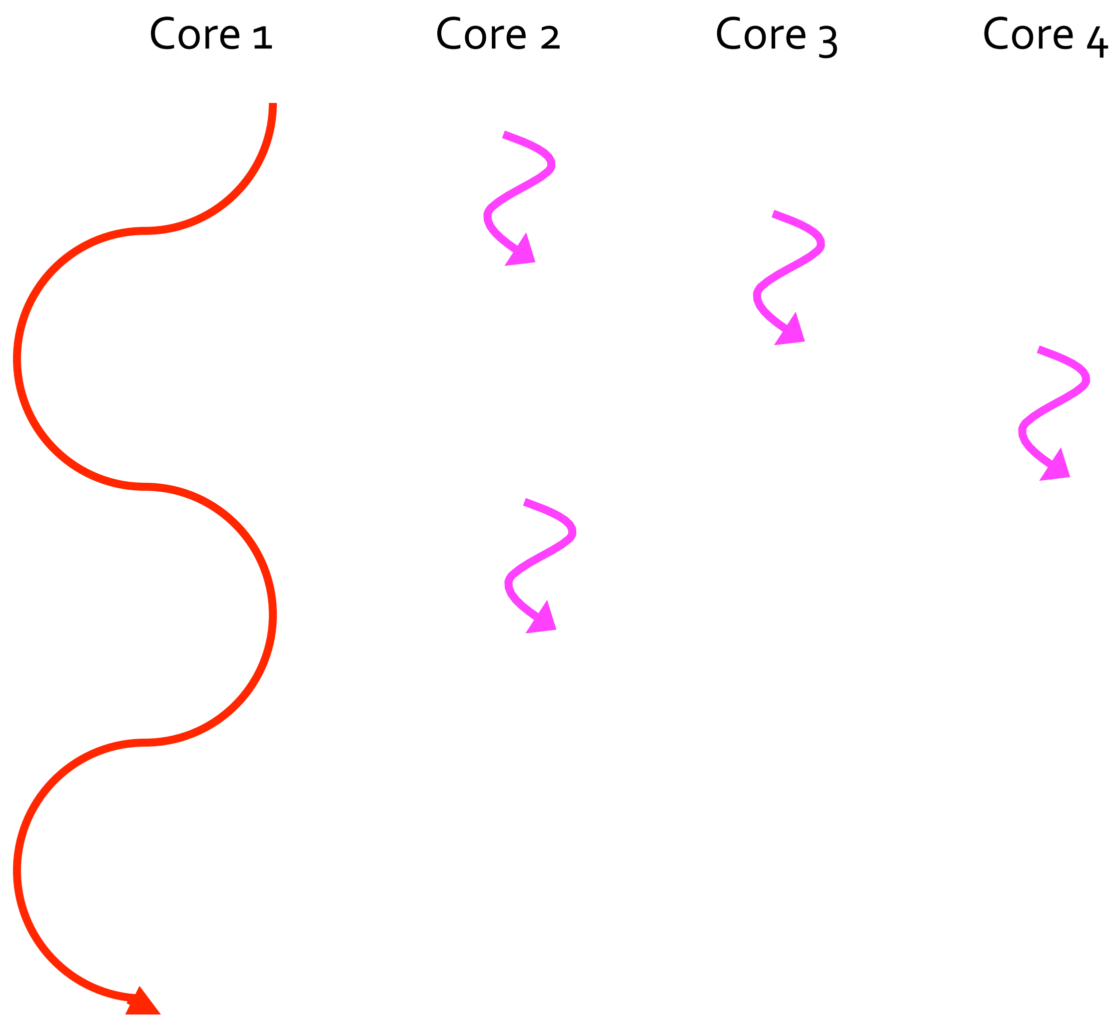
Traditional Parallelism



Traditional Parallelism



Non-Traditional Parallelism (one model)



Things we like about non-traditional parallelism

- nearly any code, no matter how inherently serial, can benefit from parallelization.
- Much more dynamic than traditional parallelism – threads can be added or subtracted without significant disruption.
- Not bound by traditional (e.g., linear) speedup limits. We often see 10X speedup with 2 or 4 cores.

Some examples of non-traditional parallelism

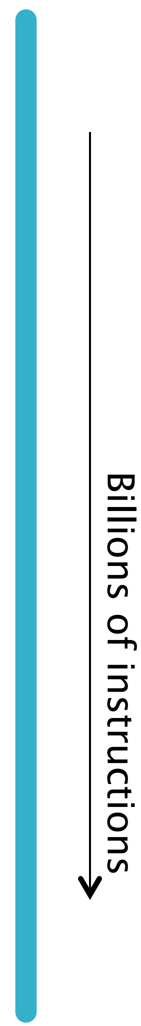
- Helper thread prefetching on multithreaded machines
- Event-driven compilation (helper threads improve code and specialize for runtime conditions)
- Software data spreading
- Inter-core prefetching
- Speculative multithreading/thread level speculation??

Some deep reservoirs of untapped parallelism

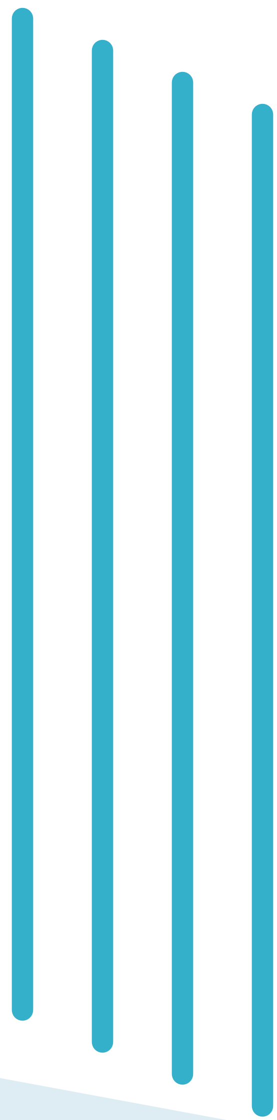
- Parallel speedup of sequential code
- Small pockets of parallelism
- Unpowered transistors

Traditional CPUs

- Optimized for:

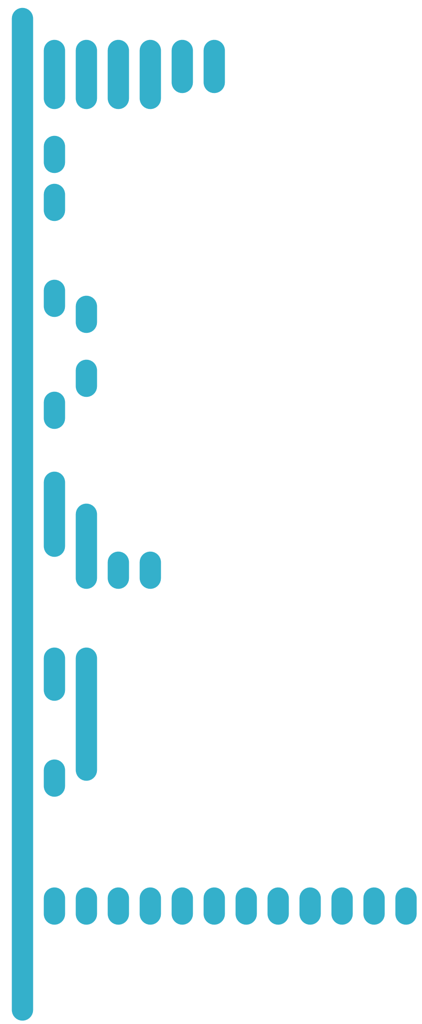


When parallel code looks like:



- Our traditional CPUs work great.

When parallel code looks like:



Then

- We find that we have the wrong
  - CPUs
  - Interconnect
  - Memory Hierarchy
  - Branch Predictors
  - Etc.

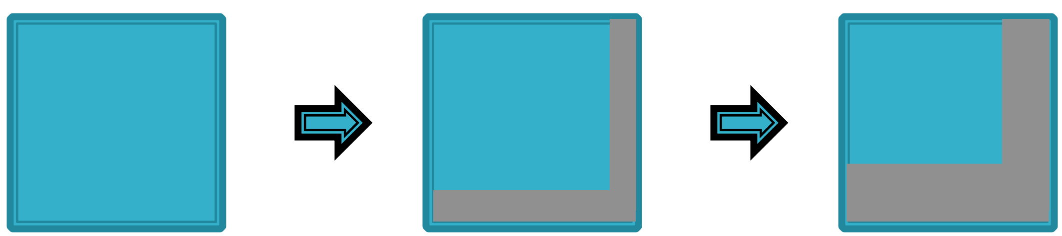
- They're all optimized for running billions of instructions without interruption. They perform very poorly when running 100s of instructions.

Some deep reservoirs of untapped parallelism

- Parallel speedup of sequential code
- Small pockets of parallelism
- Unpowered transistors

Parallelism and Dark Silicon

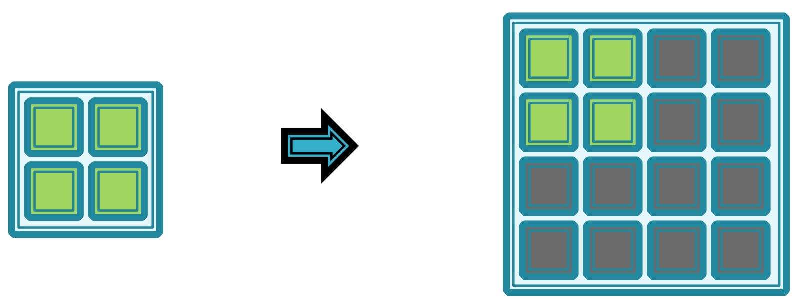
- The big point – it's easy to add transistors (cores), difficult to add more *powered-up* cores.
- Assuming expected scaling trends, larger and larger portions of the processor must remain unpowered (idle).



The Dark Silicon Question

- General: How do we add transistors/logic to the processor that add value even when they are not turned on?

- Specific to today's topic: How do we get higher parallel speedup from n cores (out of P*n total) than we can get from n cores (out of n total)?



Answers we know today…

- Heterogeneous cores. Customization, specialization…

Algorithms and Theory?

- If your models can’t explain the parallel speedups we’re achieving, then they are of limited usefulness.
  - Parallel speedup of sequential code
  - Accurately accounting for overheads of spawning computation, including sw overheads, communication, cold start, etc.
  - Handling highly irregular opportunities for parallelism
  - Accounting for power and energy bounds on computation.
  - Smoothly handling heterogeneous computing elements.

Thank you