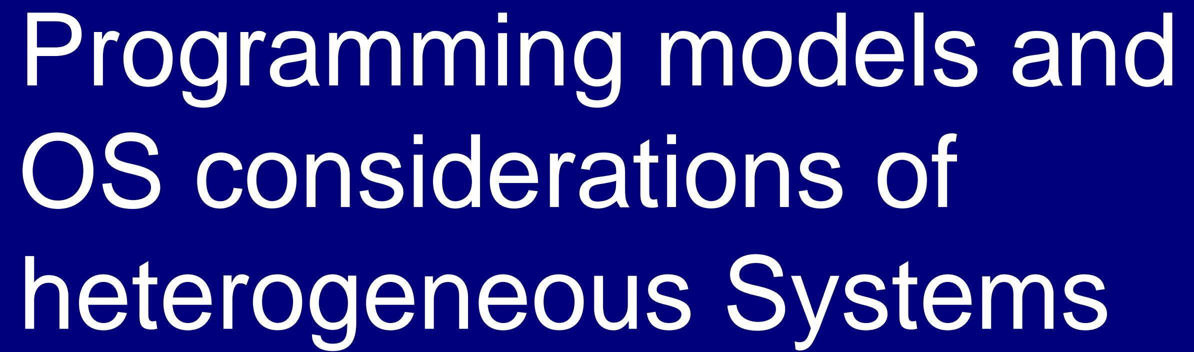


Avi Mendelson - Intel

Why we decided to “go parallel”

- We need to provide more performance in order to keep prices and value
- Couldn’t scale frequency anymore
- Observe: as long as enough parallelism exists, it is always more efficient to increase the number of cores than to increase frequency in order to achieve the same performance.

Deja-vu -- we have been there before

During the late 80’s and during the late 90’s

- Similar reasoning
- Too many companies got bankrupt as a result of these ideas.

What stopped the previous “parallel wave” ?

- Software

So now we are 20 years more experienced

- Do we know better how to program large scale TLP systems?
  - NO
- Do we know how to program DLP based systems
  - YES
    - Cuda can be used for streaming
    - OpenCL is being developed to help

But it is still HW oriented and very difficult to master

Heterogeneous systems brings a lot of fun

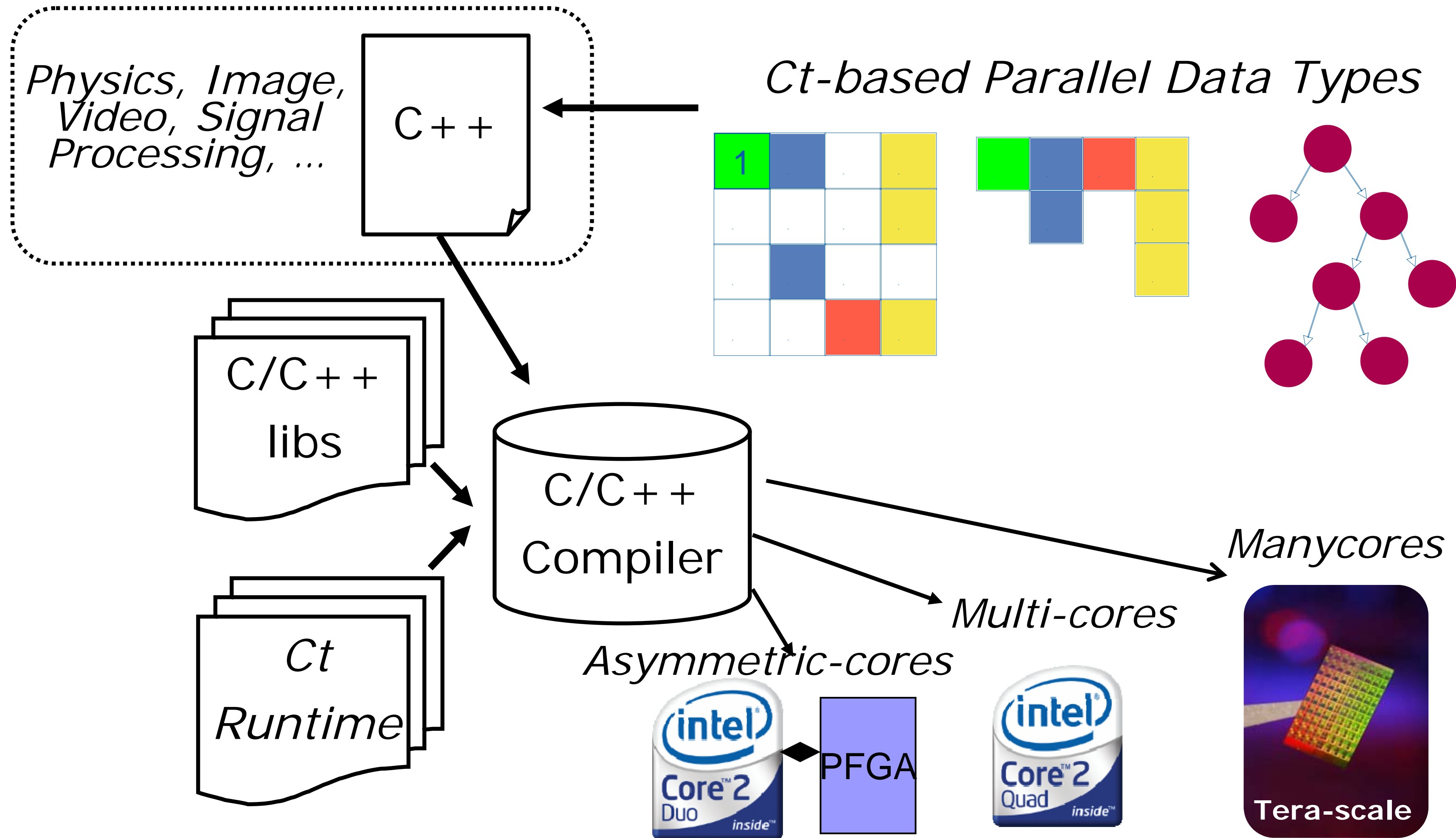
- HW perspective
  - Different subsystems may have the same ISA, or not
  - They can be controlled by the same OS, or not
  - But can save a lot of power, or not
- OS perspective
  - Same OS control the entire system, different OS control different subsystem
- SW
  - If we struggle to program multi-core systems, how can we expect to program heterogeneous system?

Asymmetric programming models that may work

- For special purpose (e.g., streaming) we have reassemble solutions
- For GP: SW can handle asymmetric cores, as long it looks like "serial" programming with special extensions such as vectors, strings manipulation etc. and appropriate SW (e.g., compiler) layer that can "extract" this parallelism
  - Users of GP programs do not like to deal with heterogeneity
- HW coherency can help a lot but may be very costly
- OS needs to be considered as a first class citizen to mask the complexity of the HW

Ct-example: Nested Data Parallelism in C/C++

Ct is a new programming language and environment, proposed by Intel research and aim at many cores and asymmetric cores.



Behind the scene: Dataflow is back!

One way of looking at Ct:

A declarative way to specify complex task graphs

What we needed:

- Fine-grained concurrency and synchronization support
  - A bunch of lightweight tasks arranged in a dependency graph
- Novel optimizations and usage patterns
  - Reuse of task graph (called *future-graph*)
  - Incremental/adaptive update of FG

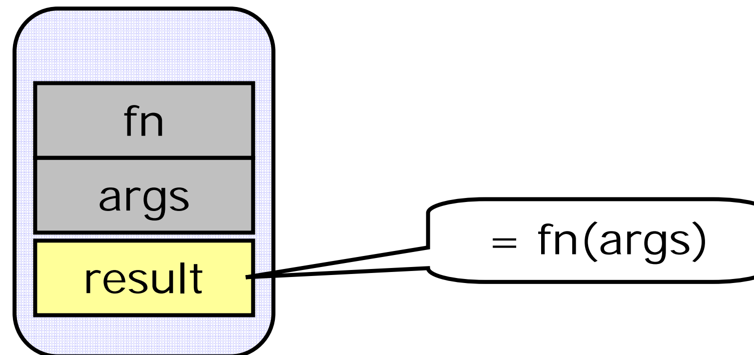
What we came up with:

- A super-lightweight *futures*-based threading abstraction
- Primitives for bulk creation of futures and complex synchronization

  → *Building blocks for dataflow-style task graphs*
- Composable first-class objects to enable dynamic optimization

Feather-weight "Threads": Futures

Futures: (Almost) stateless task



- API: Spawn & Read
- Futures can be in one of 3 states
  - Unevaluated: can be "stolen" or evaluated by reader
  - Evaluating: reader should wait for the result
  - Evaluated: reader can just grab the result
- Scheduled using distributed queues
  - Enqueued futures serviced by underlying worker threads
- Futures-creation about 2-3 orders of magnitude less expensive than thread creation

Questions?