GPGPUs in HPC

VILLE TIMONEN

Åbo Akademi University

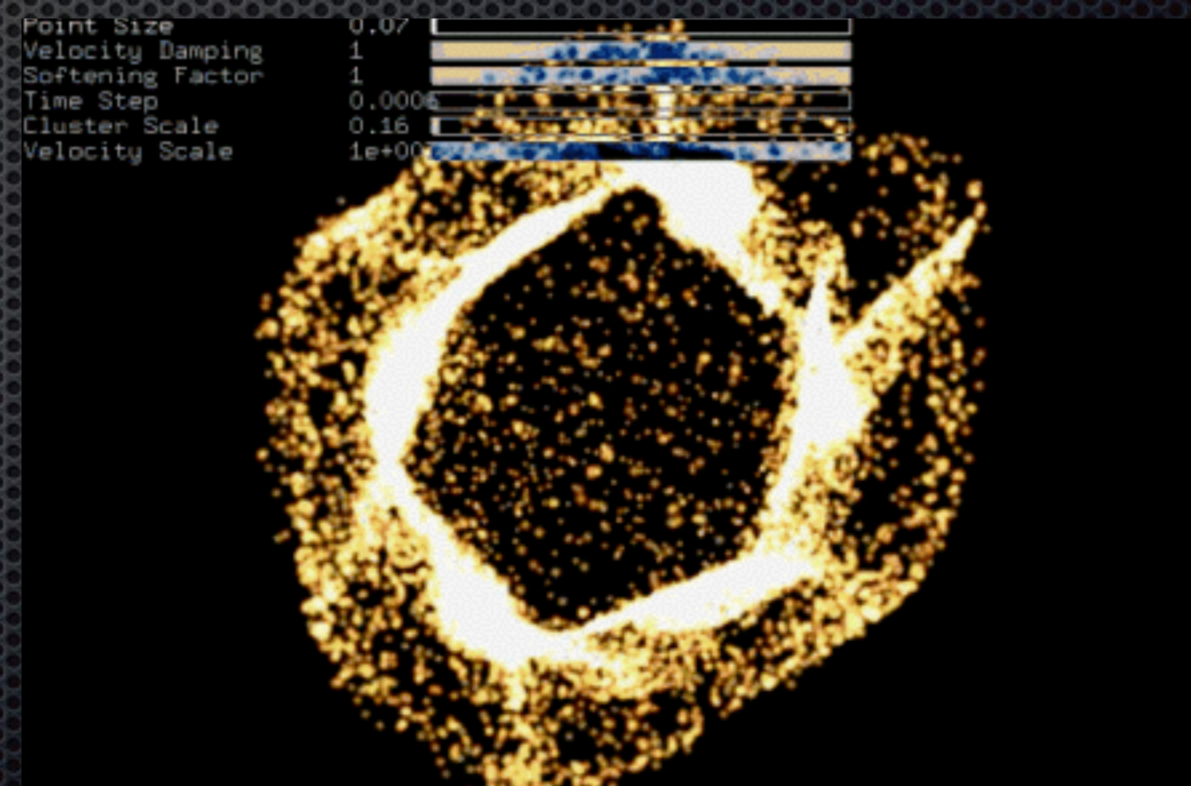
2.11.2010 @ CSC

Content

- Background
- How do GPUs pull off higher throughput
- Typical architecture
- Current situation & the future
- GPGPU languages
- A tale of one algorithm
- Conclusion

Background

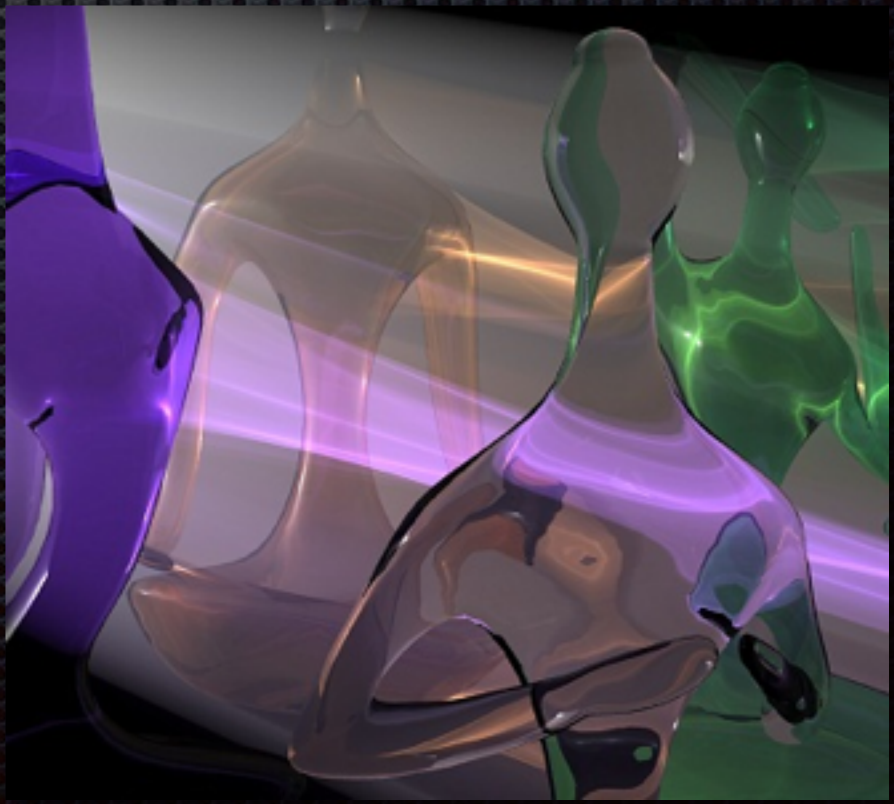
- Before GPGPU* "broke through", people mapped physics problems as graphics operations
- Programming using e.g. OpenGL shaders
- The specialized graphics HW was efficient for such operations



*) General-purpose computing on graphics processing units

Background

- On the graphics programming front, graphics libraries' abstractions are constantly being relaxed
- The usual HW is not limited to these abstractions and capable of much more
- -> Graphics is also asking for something that exposes the actual hardware better



Background

- The hardware had been ready..
- So there was need for GPGPU languages:

Graphics → GPGPU ← General-purpose

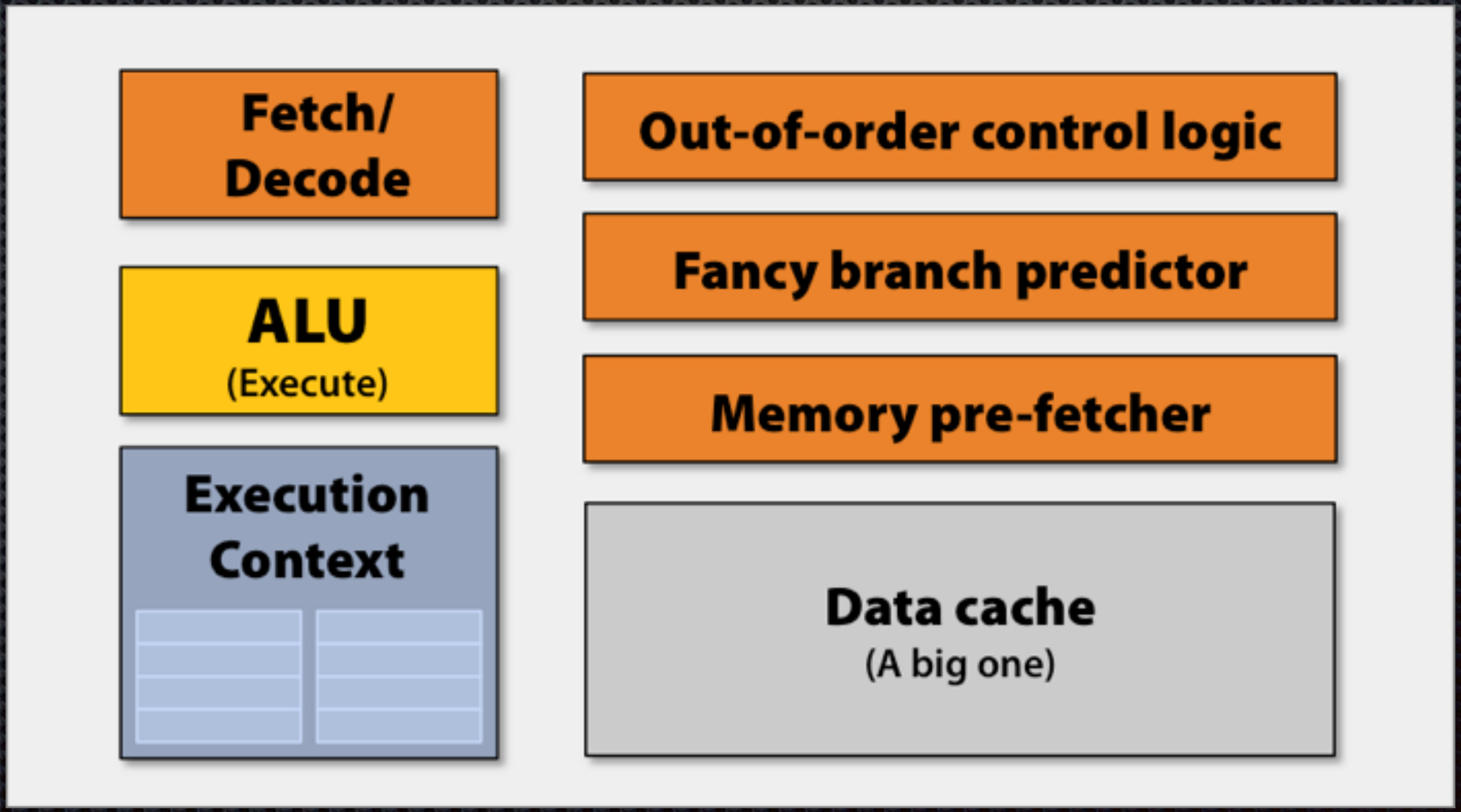
Graphics wanted more freedom with the cost of performance (optimized pipelines)

General-purpose computation wanted more performance with the cost of algorithmic freedom

How do GPUs pull off higher throughput

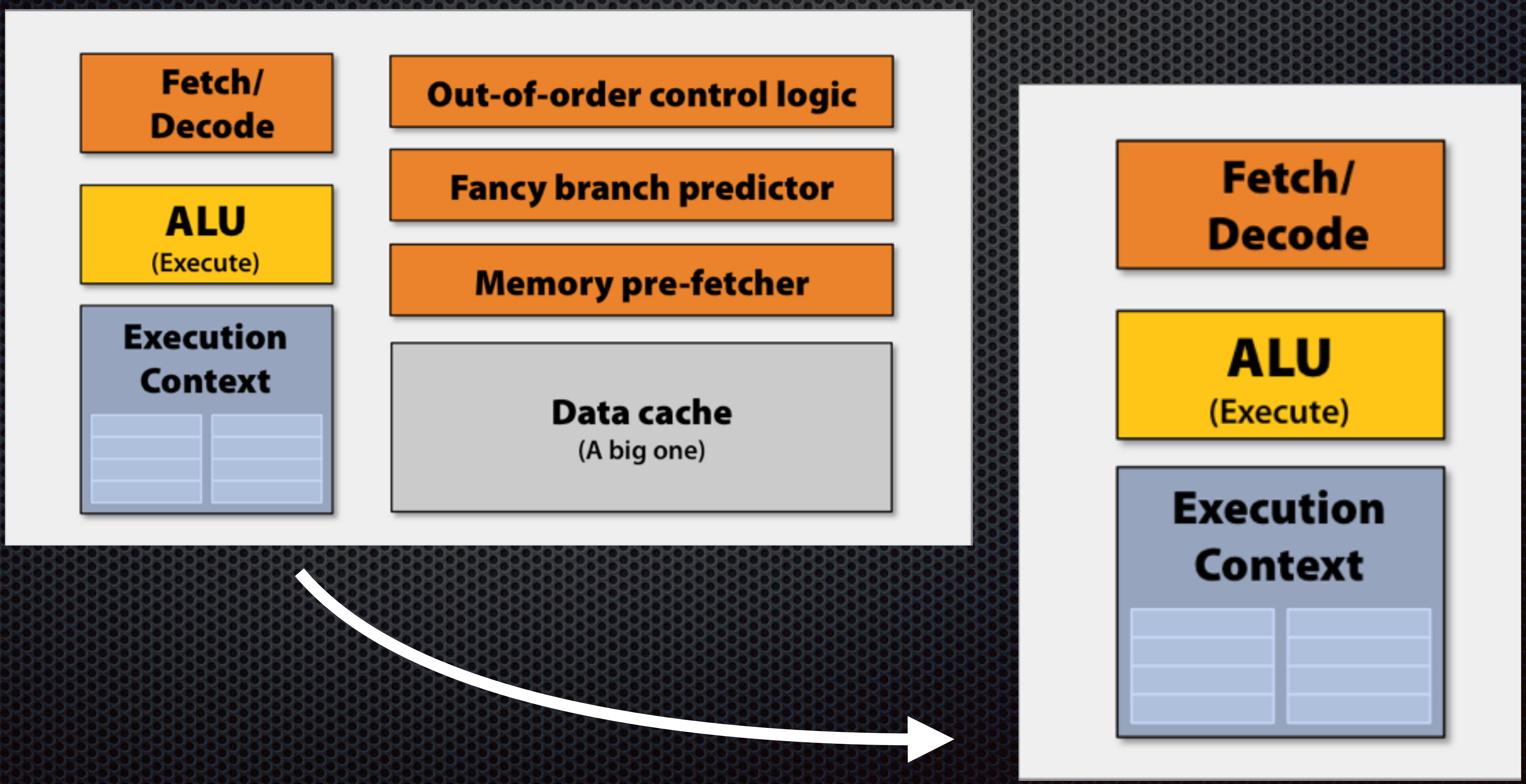
How do GPUs achieve high perf.

- It is important to understand where this performance comes from
- Let's begin by taking a commonly known CPU:



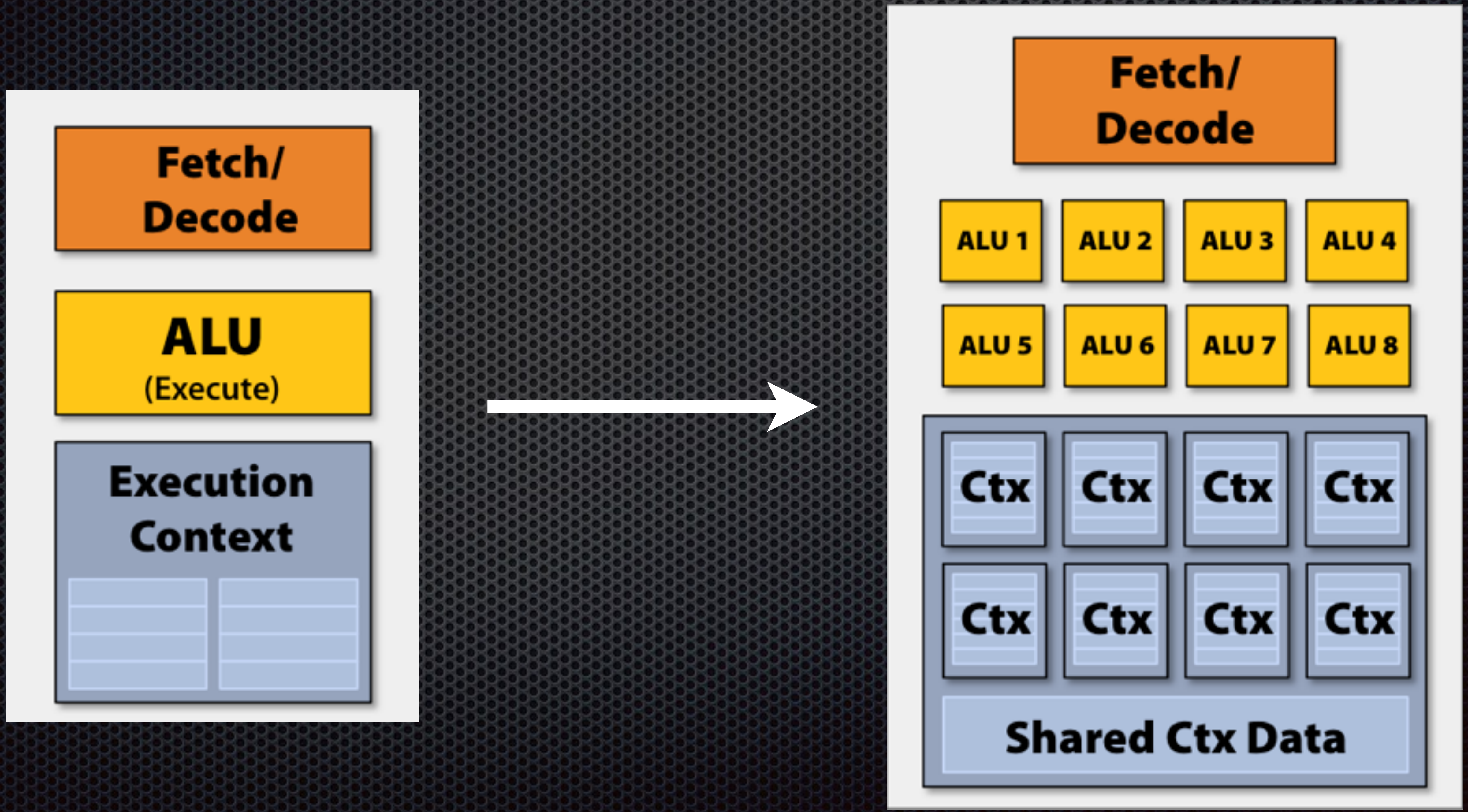
How do GPUs achieve high perf.

- Remove logic that tries to keep the exec. units busy
- And reduce the size of the cache :-(



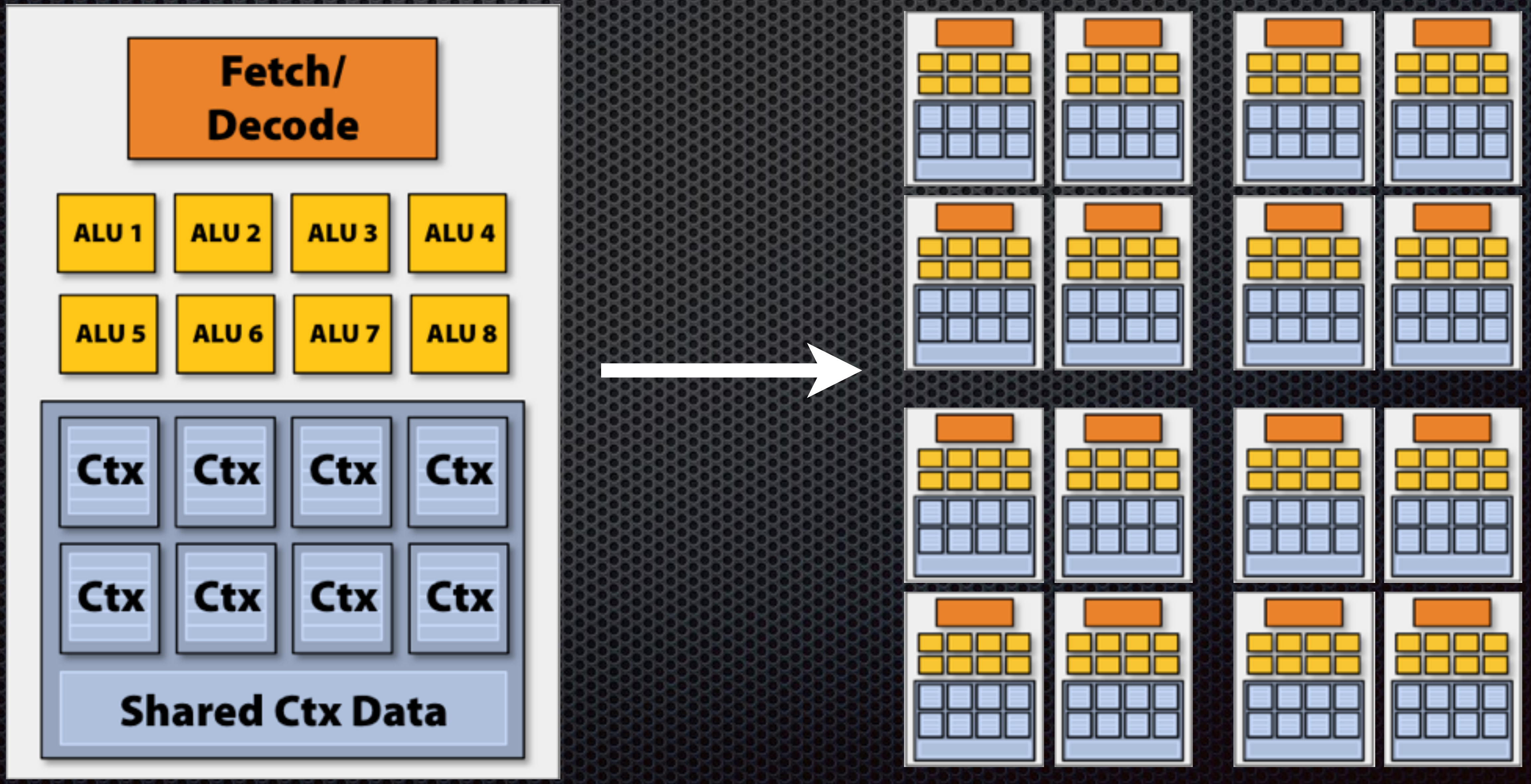
How do GPUs achieve high perf.

- Expand the SIMD units to process even more elements at once, and add registers to allow going for SIMT



How do GPUs achieve high perf.

- Multiply (up to 16-30 cores)
- Pack in fast memory and a capable memory controller



How do GPUs achieve high perf.

- It’s all about how to use your transistor budget
- Summary:
  - Logic that tries to keep the execution units busy at all times is greatly reduced, and so is the cache
  - Expand SIMD
  - Make many cores
- Furthermore, GPUs have more transistors than CPUs
  - Core i7-980X (6-core) has 1.2G, a GTX480 has 3G

How do GPUs achieve high perf.

- What about these *100x* perf. improvement boasts?
  - Unfounded. Usually based on suboptimal/bloated/single-threaded CPU implementations
  - There is up to 8 times the memory bandwidth, and up to 10 times the raw arithmetic throughput in a GPU
- The extra logic in CPUs cover up for badly optimized code -> In fact easier to get good performance with CPUs

The downside

- GPUs are specialized, not for all problems:
  - Doesn't parallelize -> don't consider GPGPU
  - Parallelizes but is control-heavy -> poor perf.
  - Depends on a large cache -> not available
  - Data-intensive requiring a data set larger than 6GB -> CPU <-> GPU transfers will kill the perf.

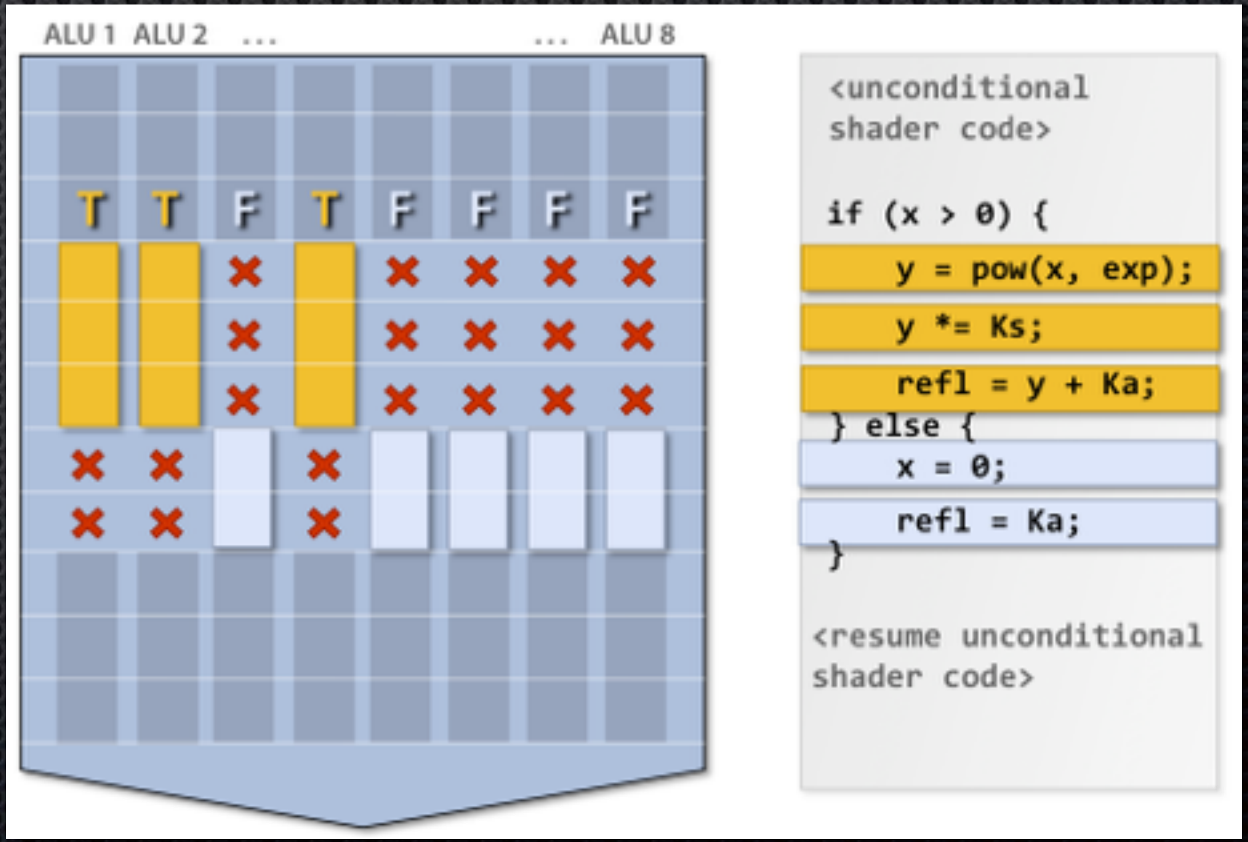
The downside

- An even larger issue is programming complexity
- Making most of a GPU requires you to have either
  (i) Very simple algorithms
  (ii) Expert GPU programming skills

Typical architecture

Typical architecture

- Issues the same instruction to multiple exec. units
- w/ CPUs, we manually packed the vectors (SIMD, e.g. SSE)
- GPUs abstract this to concurrent threads (SIMT)
- Optimal perf. only when threads agree on exec. path



Typical architecture

- Typical CPUs devote loads of transistors to logic that keeps the exec. units from stalling
- GPUs hide latencies and data dependencies by having thousands of threads in-flight to choose from
- Switching between threads essentially a no-op

Typical architecture

- GPUs have lots of registers (e.g. 128kB/core) to store contexts for so many threads
- They have small on-chip memory (e.g. 64kB/core) that can be manually accessed
  - Very little per thread, but good for communication
- Fast global off-chip memory, be careful with accessing

Typical architecture

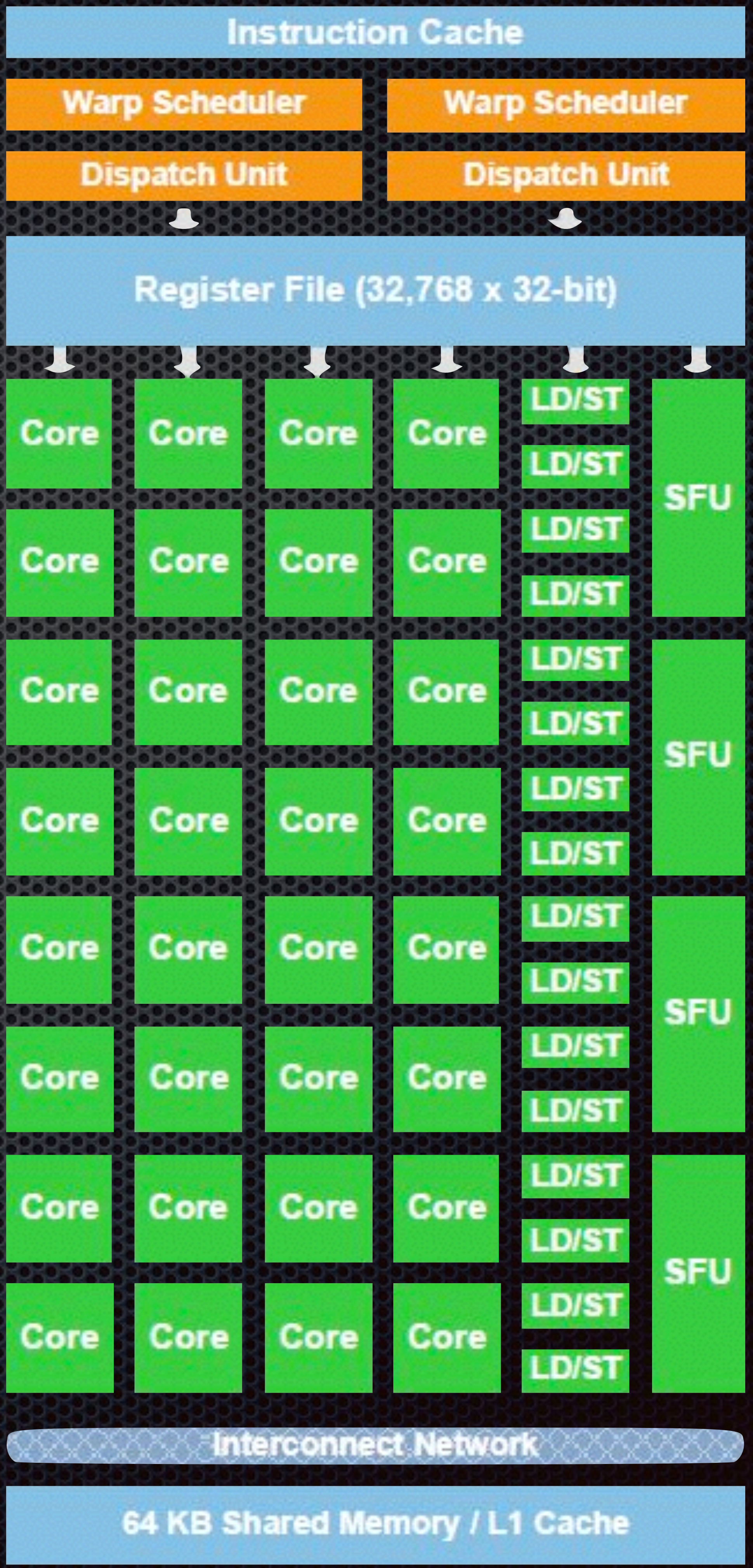
- Avoid bank conflicts with the on-chip memory
- Very important:
  - Global memory access patterns can make or break performance
  - With the exception of Fermi, there's no cache, and requests map directly to controller transactions

The Fermi architecture

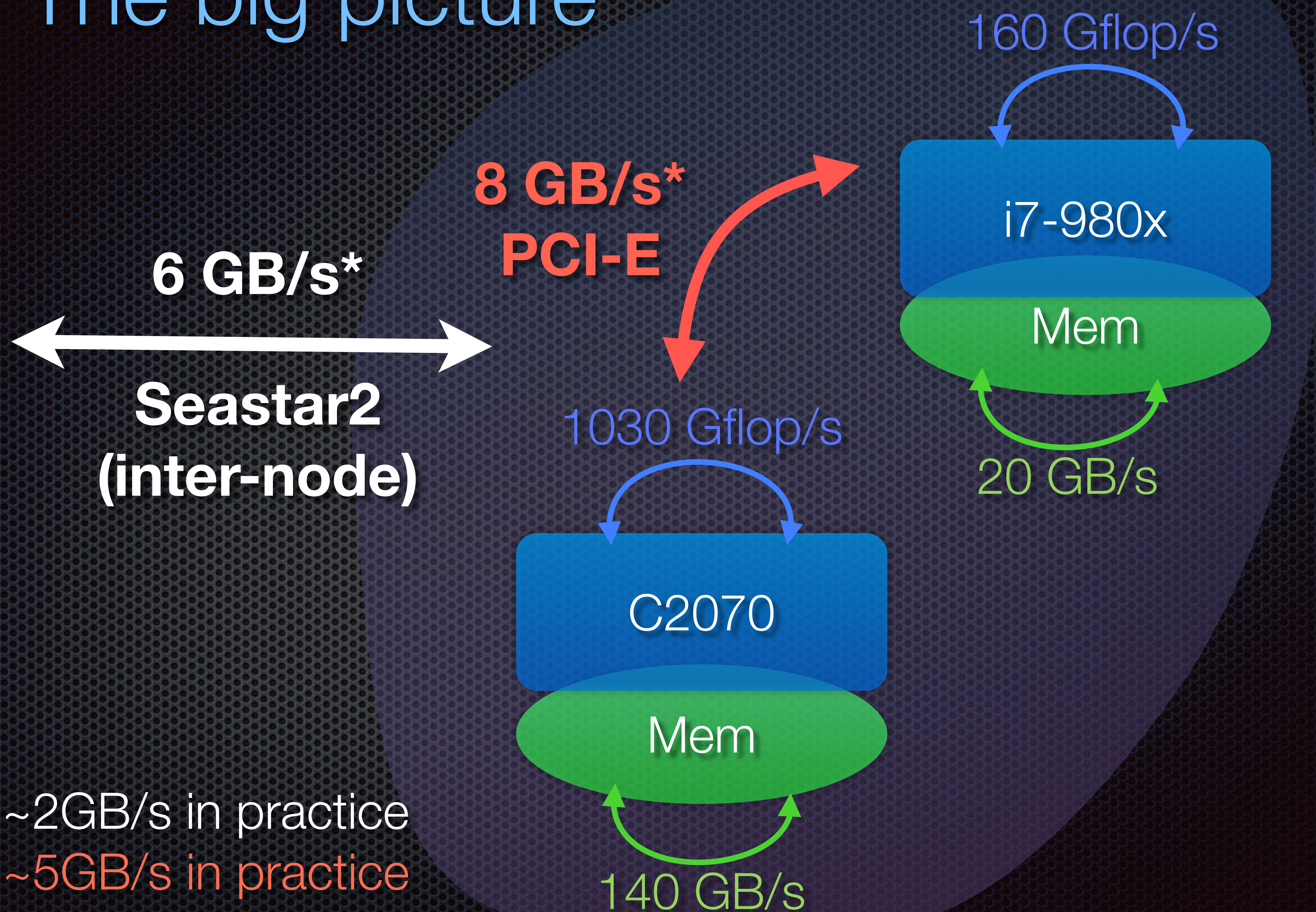
- The fastest version is the enthusiast GTX480 (1.5GB)
- Slightly slowed-down Teslas exist w/ larger memory
  - 3GB or 6GB of ECC @ 144GB/s
  - 768kB L2 cache on top of the global memory
  - 14 cores (448 exec. units total -- i7-980x has 24)
  - Up to 1 Tflop/s single precision, 0.5 Tflop/s double (i7-980x does ~140G / 70G respectively)

The Fermi core

- 32 exec. units
- Dual-scheduler
- Capable of running a kernel independently
- On-chip mem. configurable as L1



The big picture



*) ~2GB/s in practice
*) ~5GB/s in practice

Current situation & the future

Current vs. future

Throughput (Gops/s)

Specialization

ATI

NVidia

Intel

Scheduling logic

Utilization

Ease of programming

Generalization

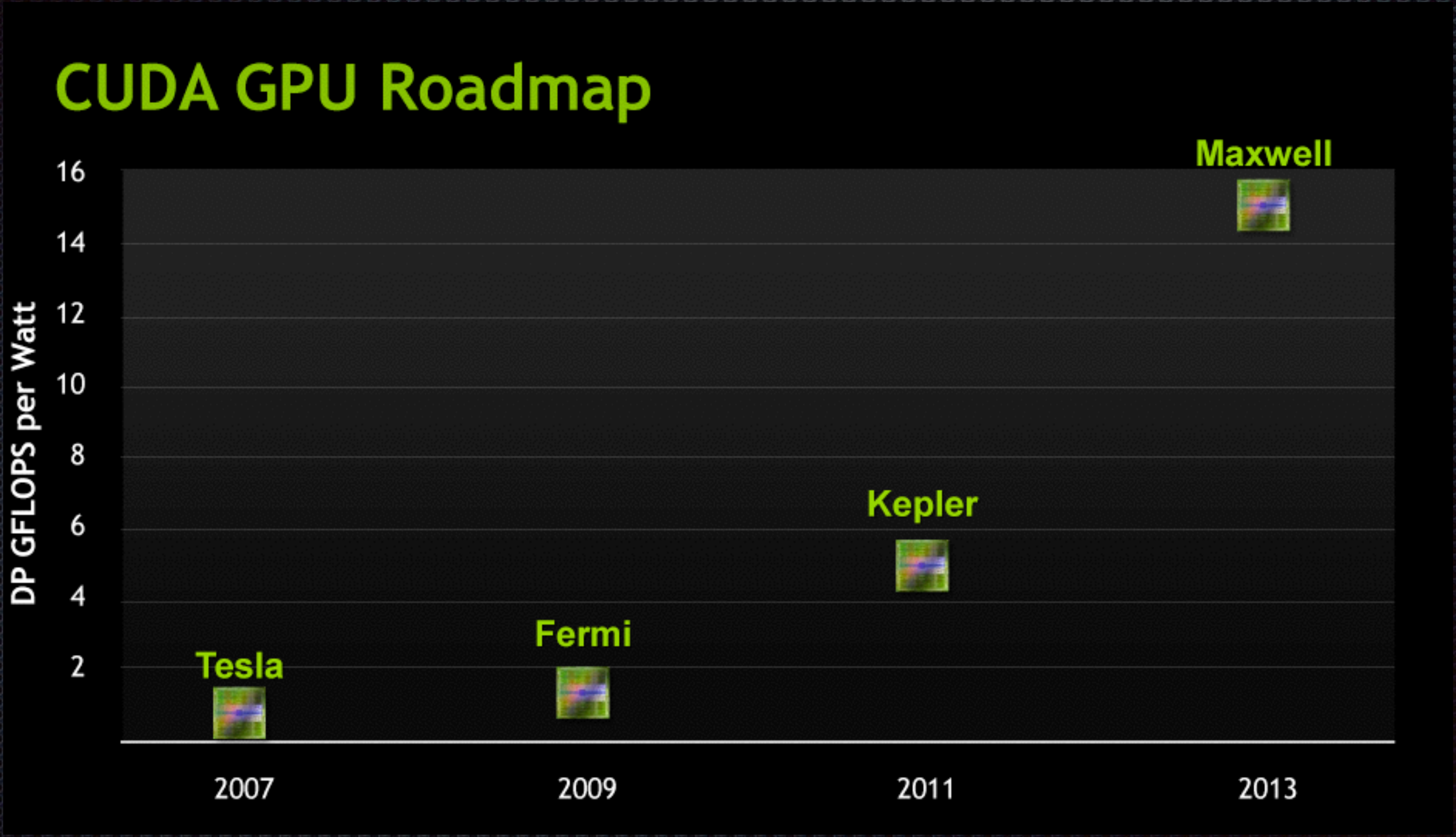
Current vs. future

- Could we combine a GPU and a CPU into a hybrid?

- They are architecturally different, that's the whole idea
- They both have global memory and a cache on top of it
  - We could probably unify these -> fast communication
    - CPU mem. is too slow currently

- We could put the extra CPU logic to only some of the GPU cores!

Current vs. future



GPGPU languages

GPGPU languages

- High level languages:
- OpenCL
- C for CUDA (NVidia)
- DirectX Compute (Microsoft)
- Cal/Brook+ (ATI, but prefer OpenCL)

GPGPU languages

- OpenCL

- Khronos's GPGPU standard
- Everyone's in it: AMD, Intel, Apple, Nokia, NVidia...*
- So it moves slowly

*) Except Microsoft of course, who wants their proprietary solutions to triumph over the open ones

GPGPU languages

- CUDA
- NVidia's solution for early adopters
- Has been around the longest, most mature
- Support for every new hardware feature NVidia releases will be immediately available
- Has optimized libraries (by NVidia) for common tasks:
  - BLAS routines (all levels)
  - Sparse matrix operations
  - Pseudo random number generation
  - FFT routines, etc...

GPGPU languages

- Choose OpenCL whenever you can
- If you do something fancy you might need CUDA
- If plan on using ready-made libs, today, choose CUDA

GPGPU languages

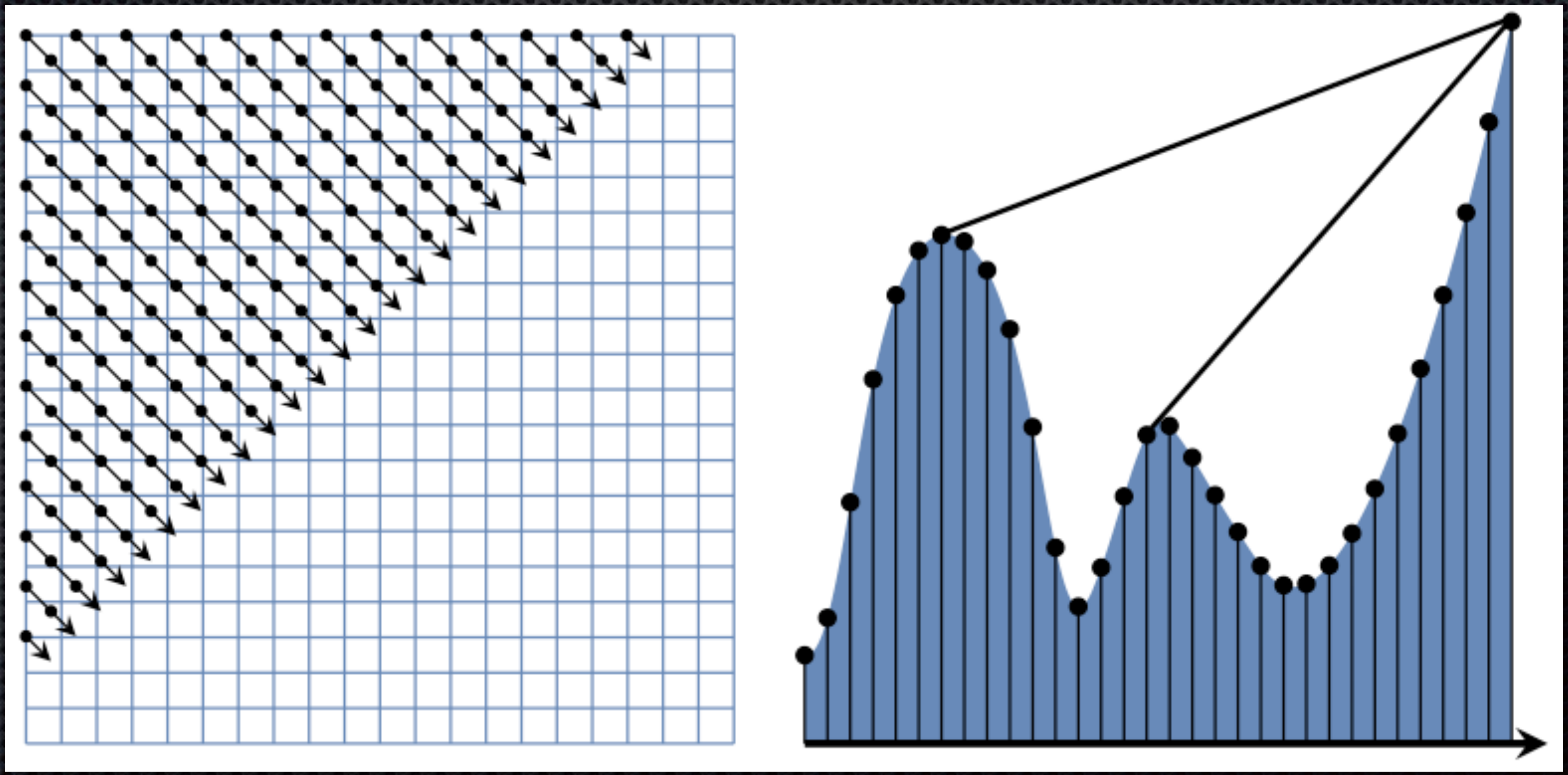
- The future?
  - Some compilers already make GPGPU code from C
    - Works decently for only the simplest of algorithms
    - Proper porting is still an expert task
  - The existing GPGPU libs/toolkits are easy-to-use
    - Do enough ops per data to avoid PCI-E bottleneck
      - Supercomputer network < PCI-E

A tale of one algorithm

A case: my latest algorithm

A case: my latest algorithm

One thread per one line



A case: my latest algorithm

- Recursive algorithm executed at each iteration on a tree
- Data set for one thread fits easily into CPU cache
- Achieve 3.3 Gops/s on 2.5GHz Xeon, single-threaded

A case: my latest algorithm

- GPU implementation

- Recursion was new and broken when I started
- Manual implementation w/ global memory stack

- At this point made over 10 revisions of the algorithm
  - The fastest one also used in CPU

A case: my latest algorithm

- With GPUs, 2k threads in-flight
- Cache usage becomes critical
- Spent significant amount of time optimizing
  - *5x* boost in performance

- Result:
  - 65% of accesses caught by the L1 cache
  - 35% causes 58GB/s traffic

A case: my latest algorithm

- Adjacent threads execute different amount of iterations of the “recursive” algorithm
- If one does 100 iterations, other 31 have to wait
- Measured utilization caused by this 18%

- We still achieve 71 Gops/s
  - (400 Gops/s if utilization were 100%)
  - 60% of the theoretical maximum
- It took 3 months to optimize, and I’m not new to this

A case: my latest algorithm

- A sweep took 190ms on CPU, 3.13ms on GPU
- 60x improvement
- If we extrapolate to a faster 6-core CPU, and assume linear scaling w/ multithreading, it's 8x improvement
- Still not bad for an algorithm that at first looked like a suboptimal GPGPU candidate (recursive, low utilization, not enough cache, scattered reads)

Conclusion

- GPUs
  - 3x the performance per transistor
  - Faster global memory
- The cost: removed logic
- Keeping the efficiency up is now up to the programmer

Conclusion

- Porting to GPGPUs is generally an expert task
- In my opinion will not be properly automatized in near future (except for simple loops)

- GPUs are specialized, but HPC can benefit from them
- Toolkits and external libs have optimized standard routines and are easy to use