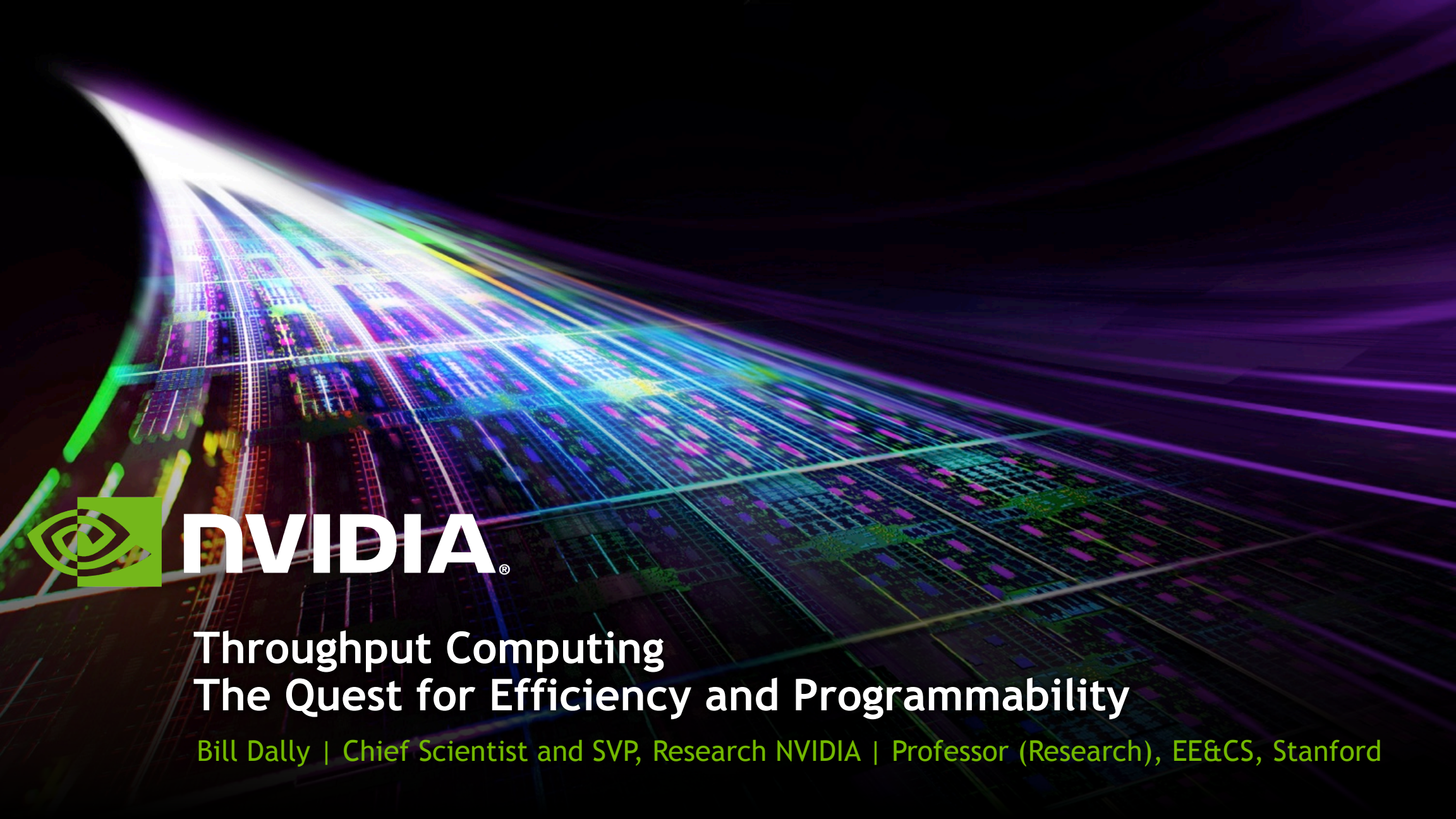


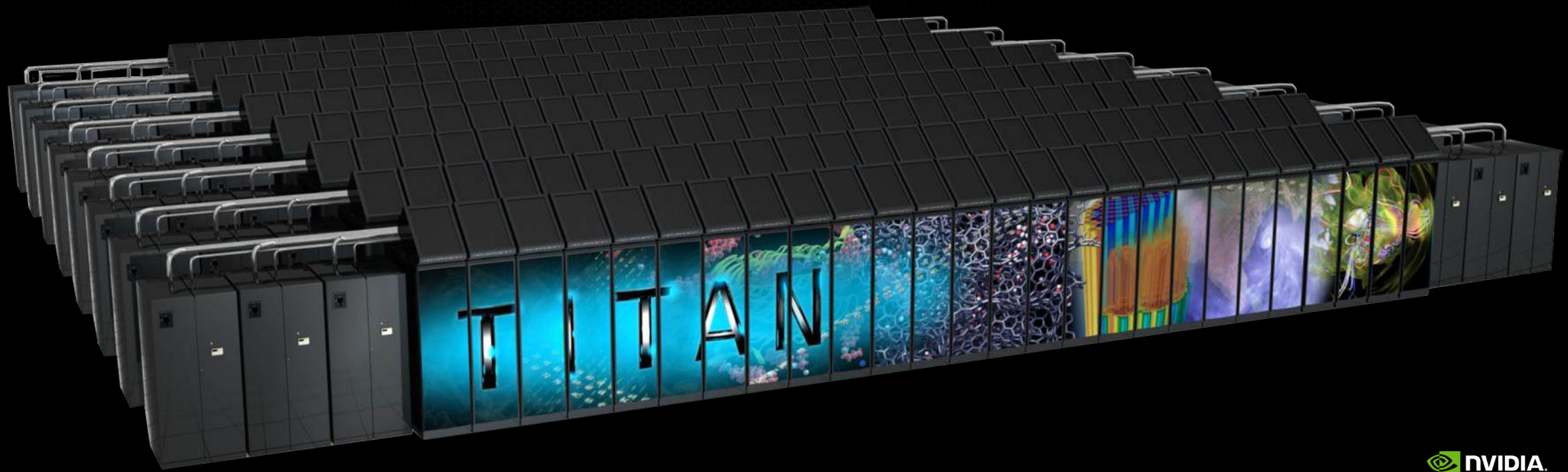
Throughput Computing
The Quest for Efficiency and Programmability

Bill Dally | Chief Scientist and SVP, Research NVIDIA | Professor (Research), EE&CS, Stanford

The Exascale Challenge

Sustain 1EFLOPs on a “real” application

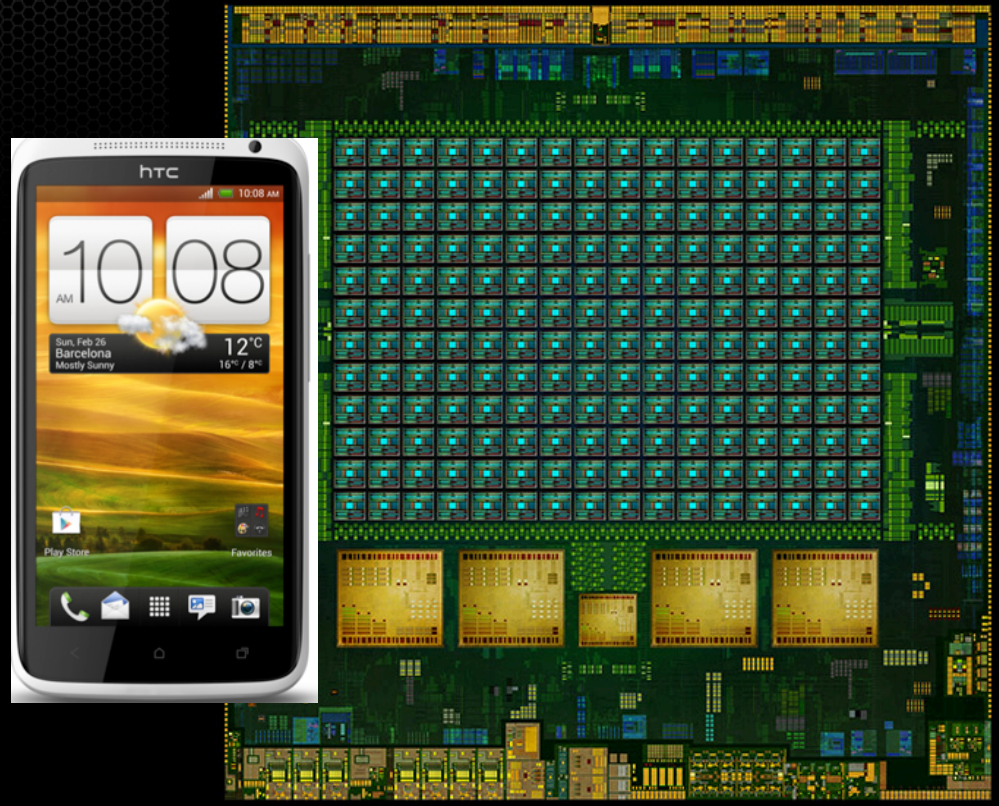
Power less than 20MW



The Cellphone Challenge

Deliver 50GFLOPs on mobile applications

Power < 1W



From cell phones to supercomputers we are *Power Limited*

Perf/W is Performance

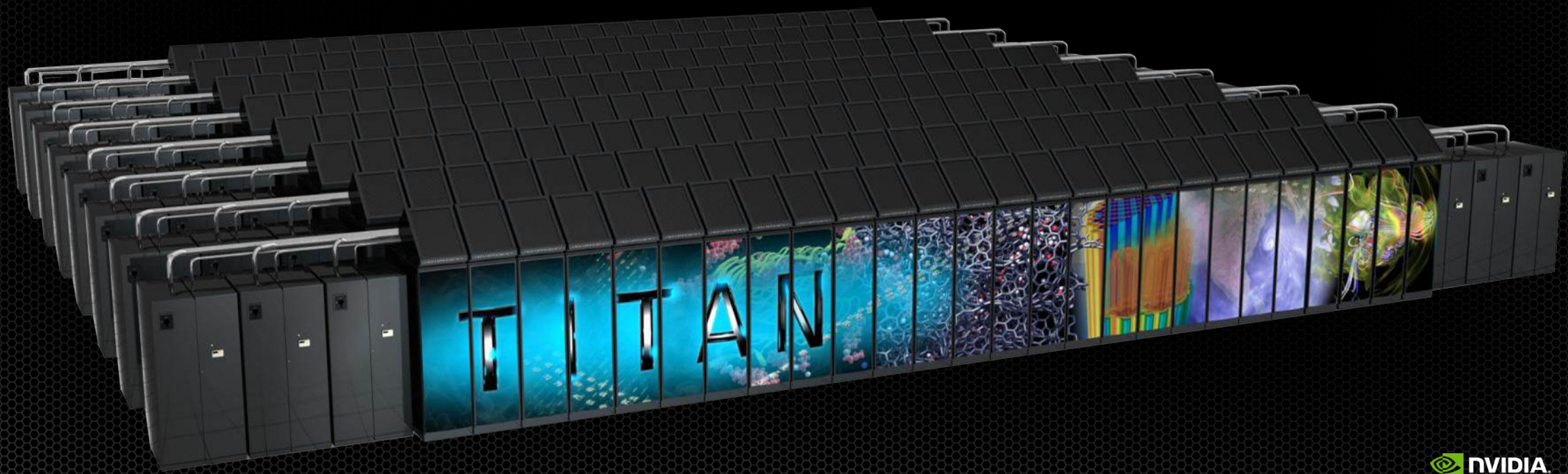
And we need to make it easy to program both these devices

TITAN

18,688 NVIDIA Tesla K20X GPUs

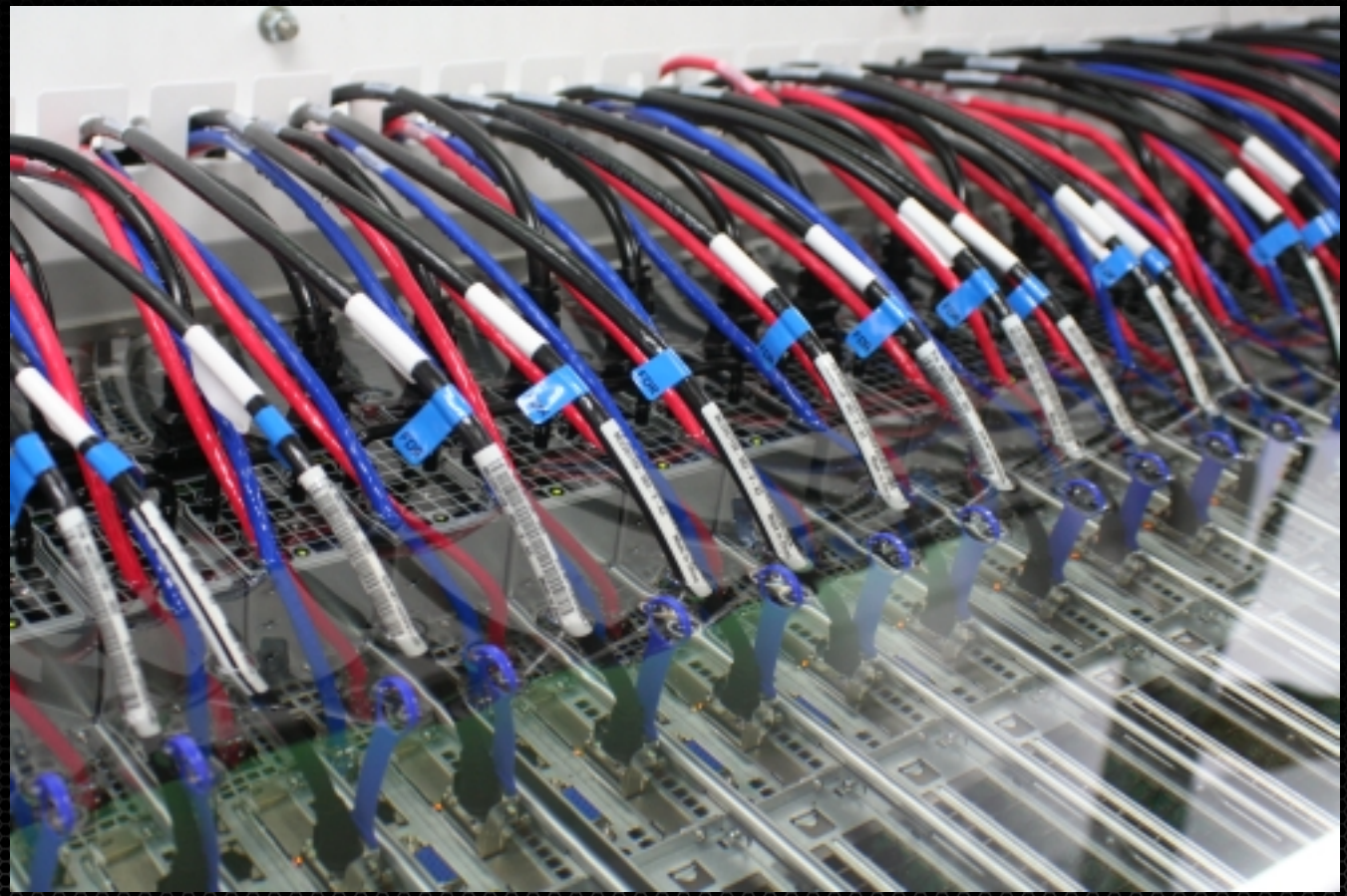
27 Petaflops Peak: 90% of Performance from GPUs

17.59 Petaflops Sustained Performance on Linpack



NVIDIA

Tsubame KFC 4.5GFLOPS/W
#1 on Green500 List



Its not about the FLOPs

- DFMA 0.05mm^2 10pJ/OP – 2GFLOPs

16nm chip, 25mm on a side, 200W

A chip with 10^4 FPUs:

500mm^2

200W

20TFLOPS

Pack 50,000 of these in racks

1EFLOPS

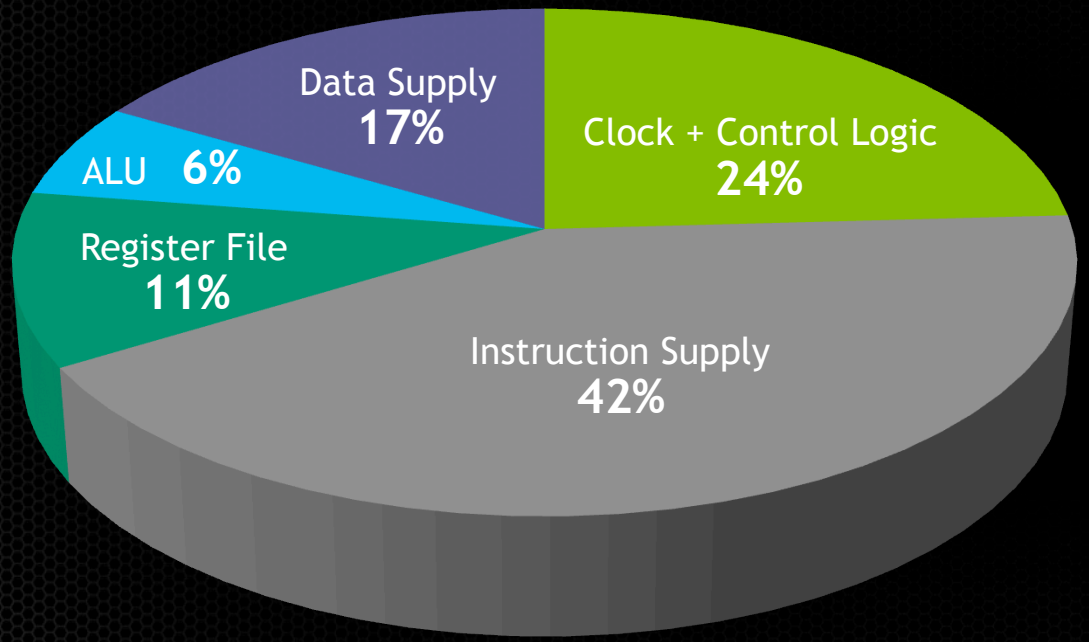
10MW

Overhead

Locality

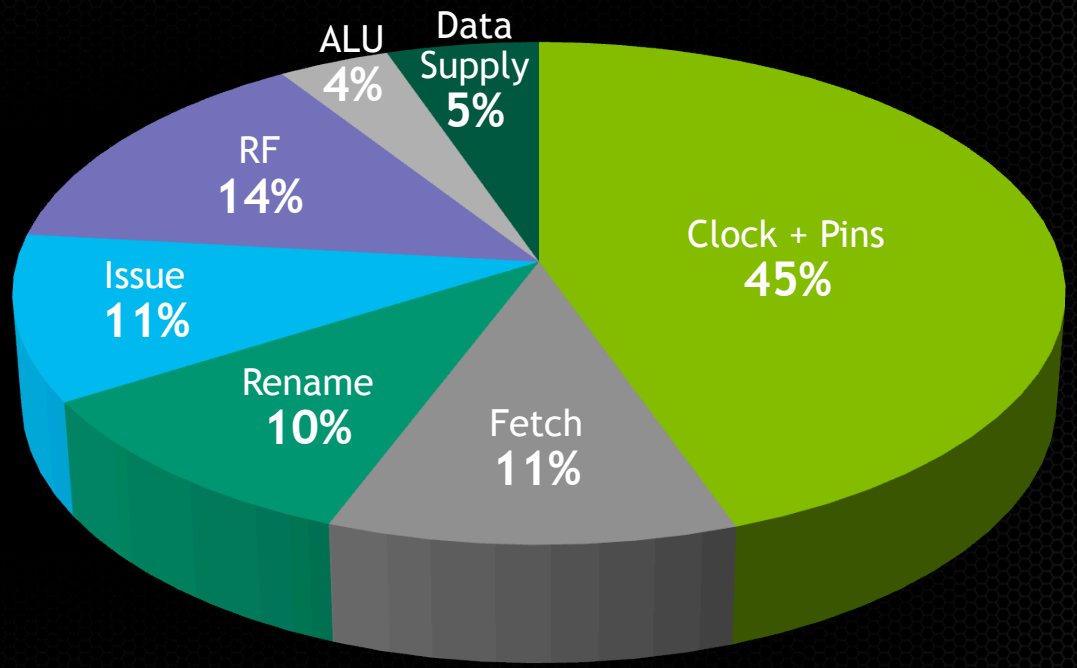
How is Power Spent in a CPU?

In-order Embedded

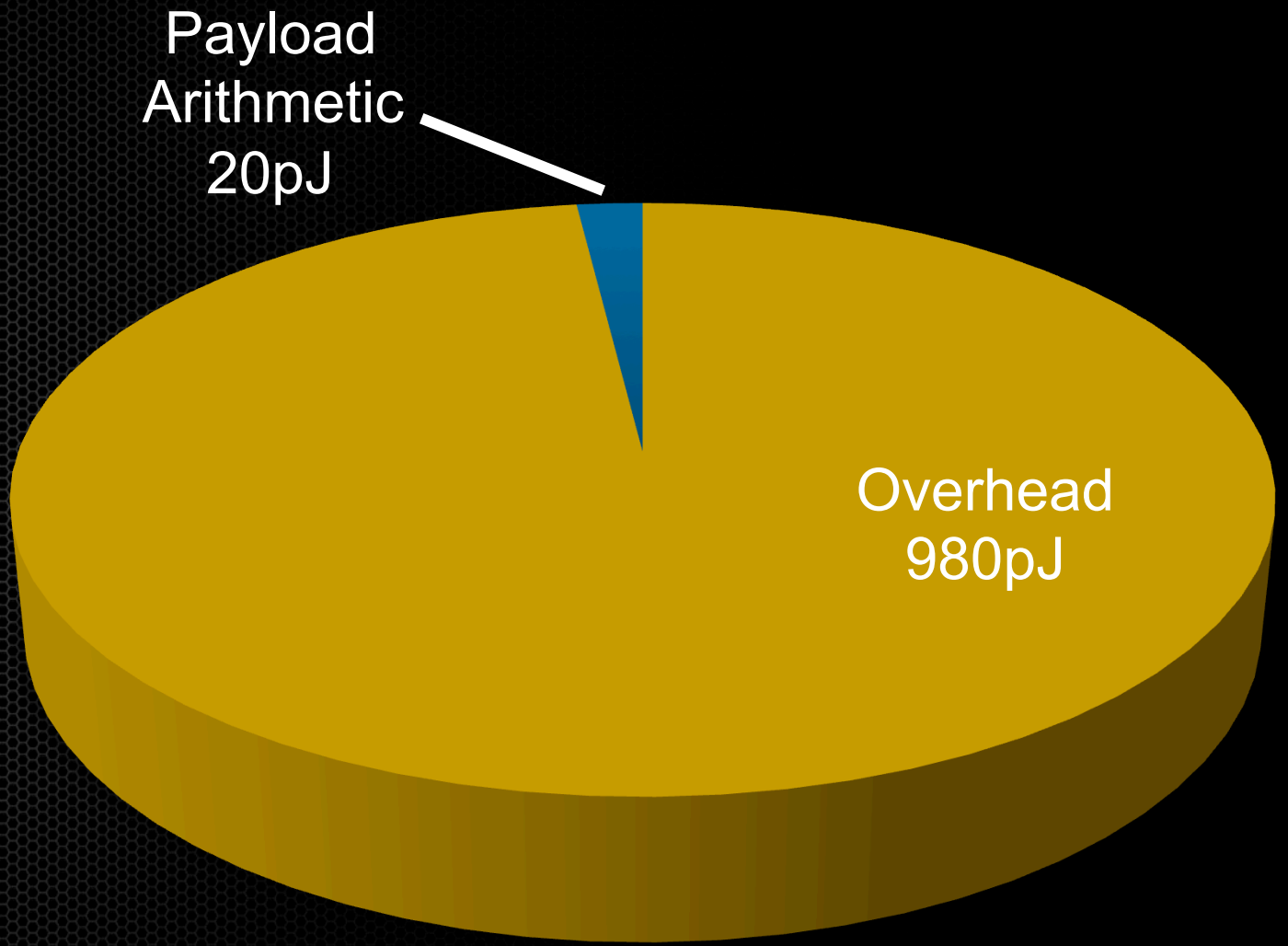


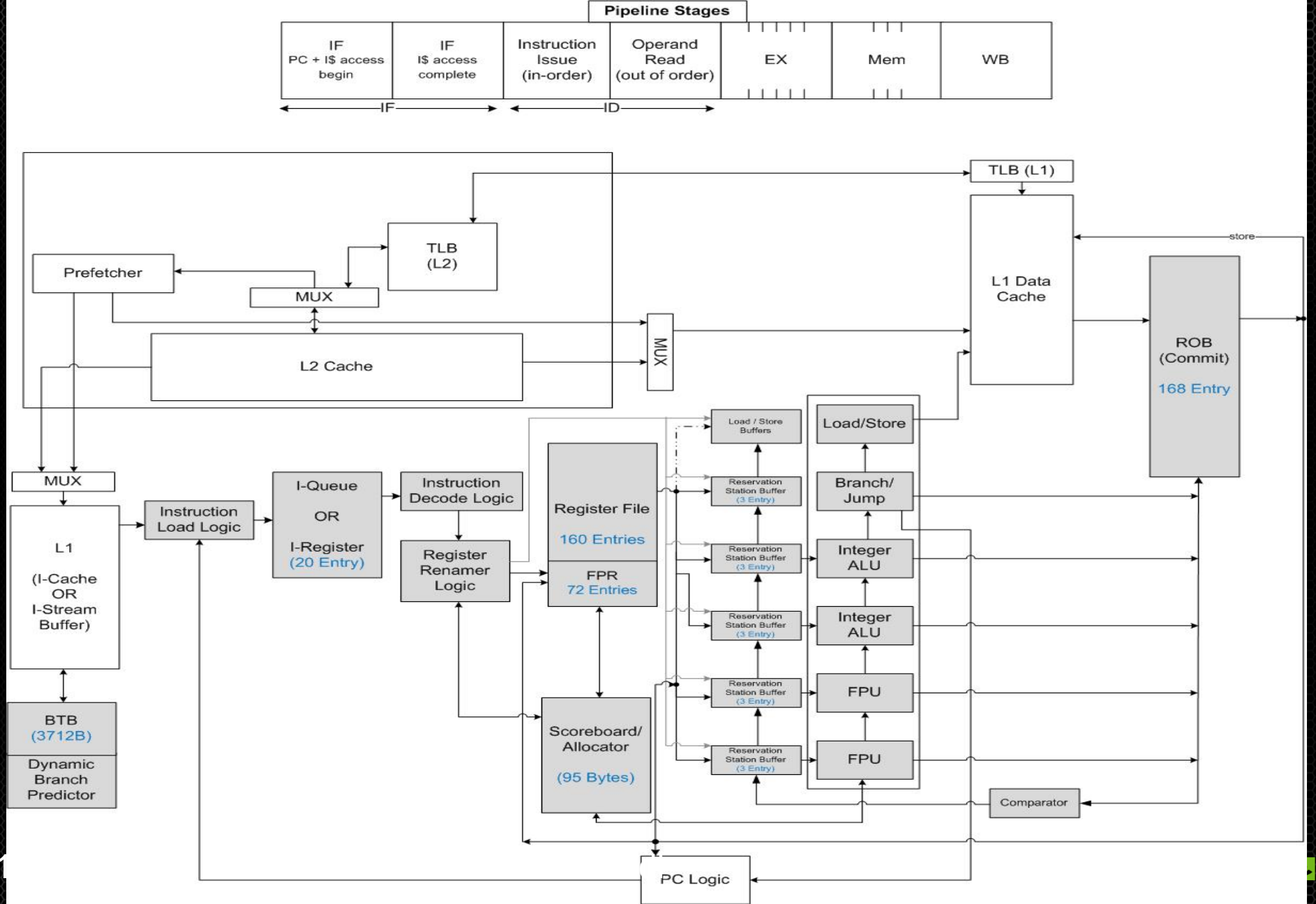
Dally [2008] (Embedded in-order CPU)

OOO Hi-perf

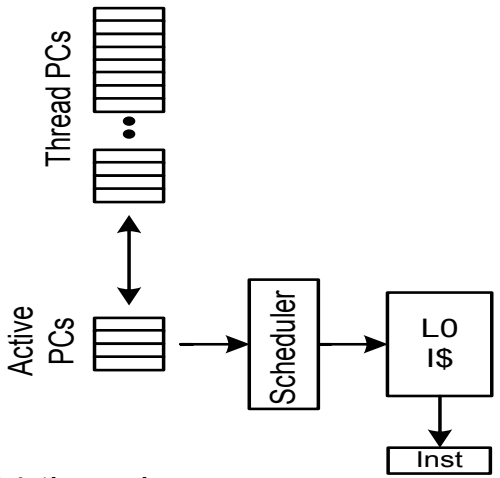


Natarajan [2003] (Alpha 21264)



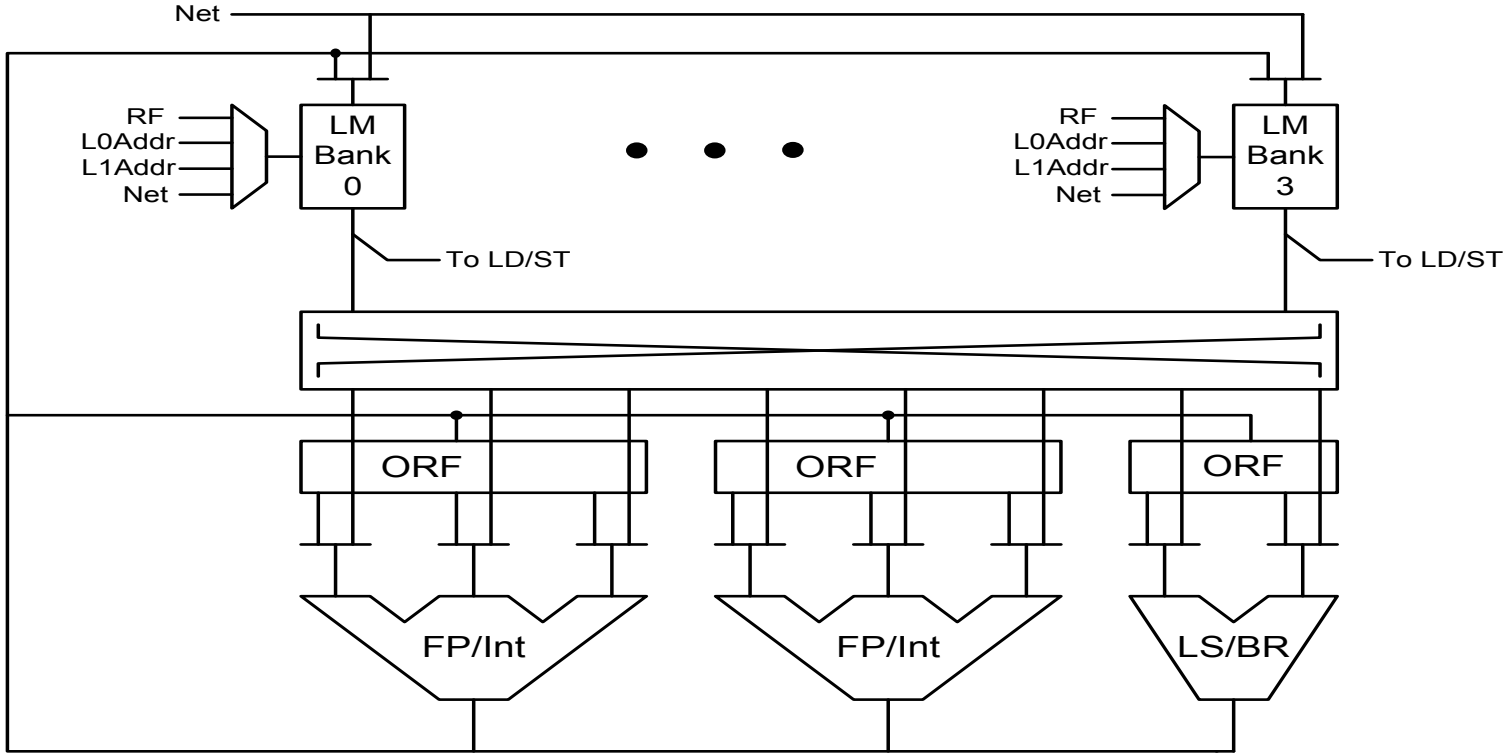

Pipeline Stages
IF
PC + I$ access begin
IF
I$ access complete
Instruction Issue (in-order)
Operand Read (out of order)
EX
Mem
WB
IF
ID
TLB (L1)
TLB (L2)
Prefetcher
MUX
L2 Cache
MUX
L1 Data Cache
store
ROB (Commit)
168 Entry
MUX
L1 (I-Cache OR I-Stream Buffer)
Instruction Load Logic
I-Queue OR I-Register (20 Entry)
Instruction Decode Logic
Register Renamer Logic
Register File
160 Entries
FPR
72 Entries
Load / Store Buffers
Reservation Station Buffer (3 Entry)
Load/Store
Branch/ Jump
Integer ALU
Integer ALU
FPU
FPU
BTB (3712B)
Dynamic Branch Predictor
Scoreboard/ Allocator
(95 Bytes)
Comparator
PC Logic

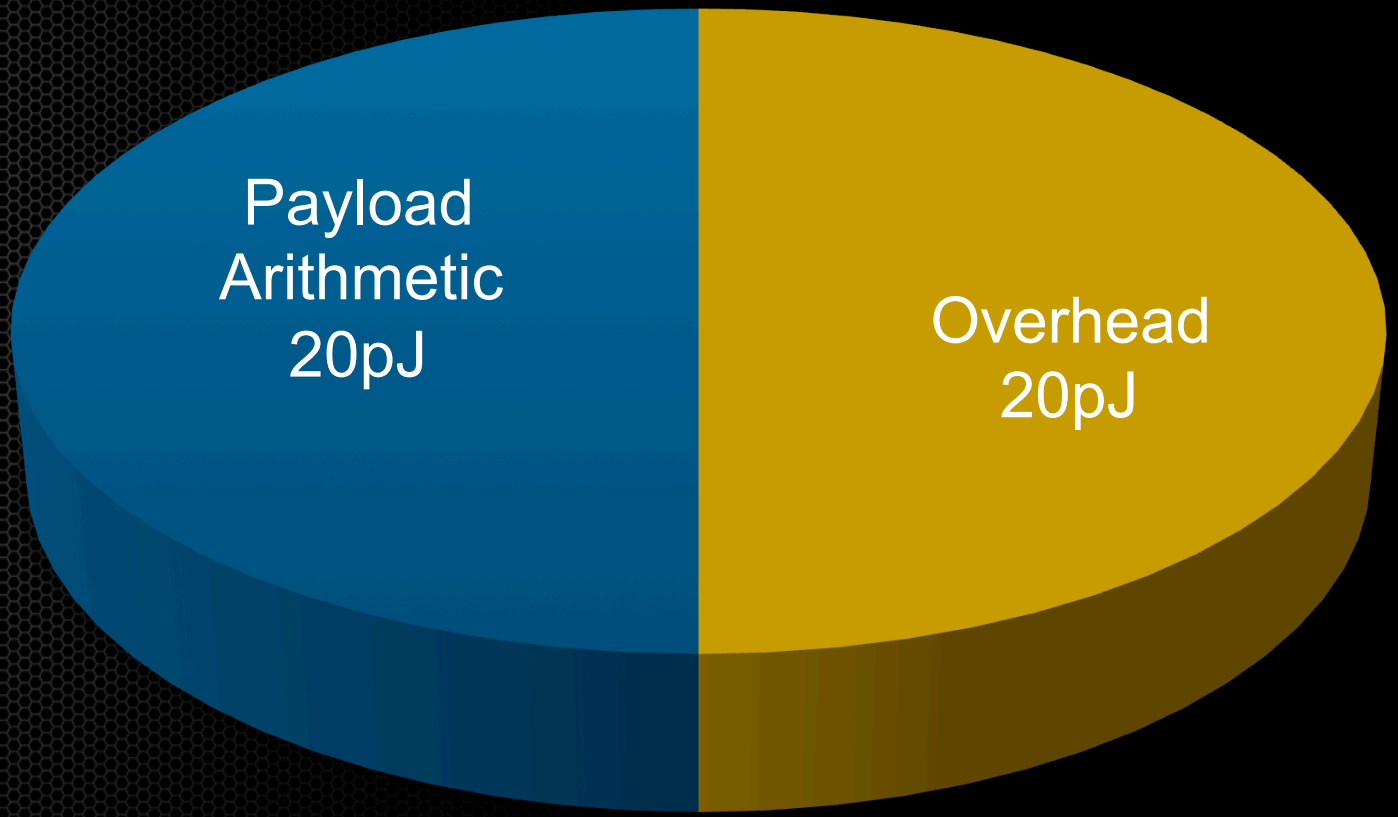
Control Path



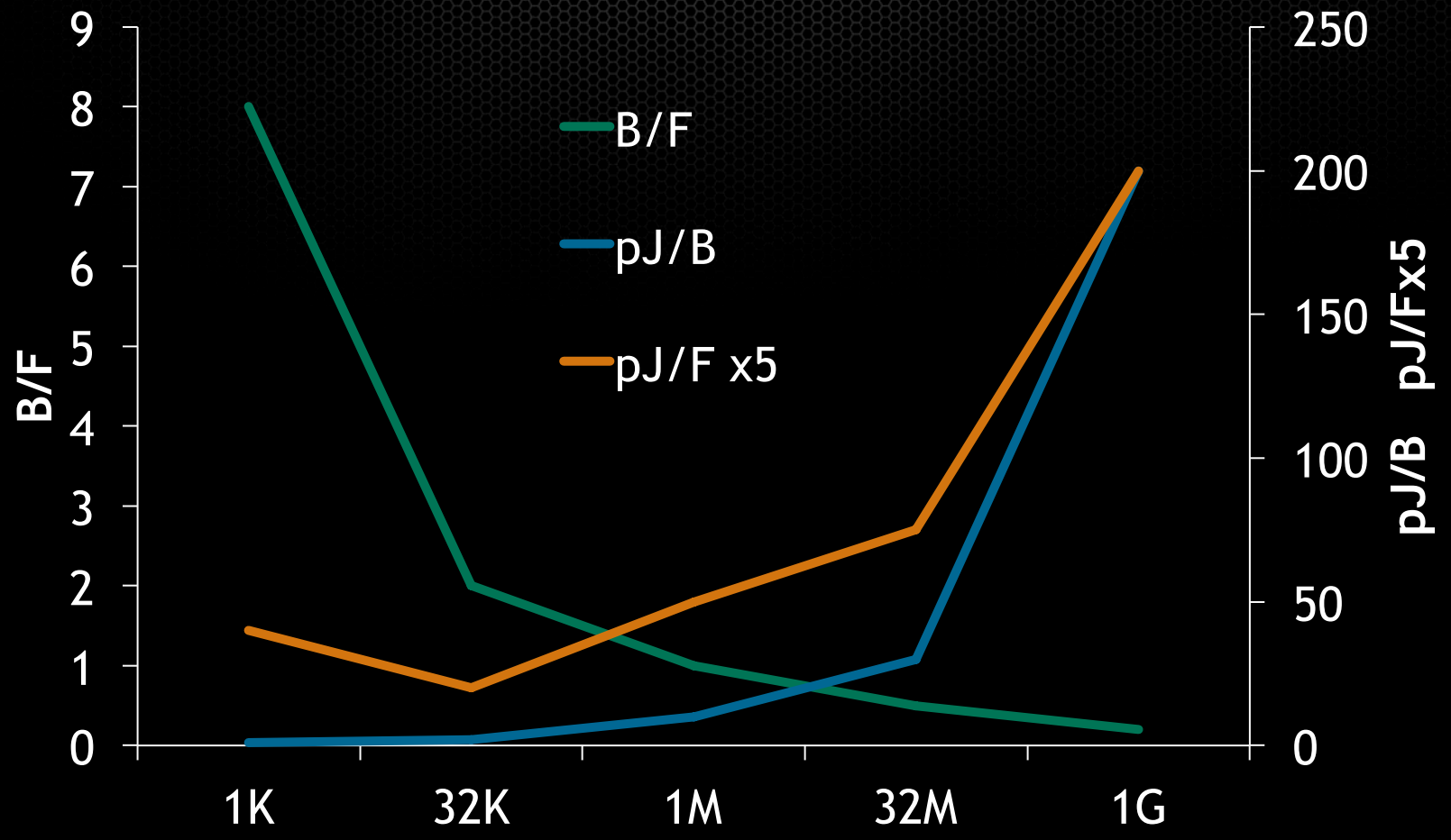
64 threads
4 active threads
2 DFMAs (4 FLOPS/clock)
ORF bank: 16 entries (128 Bytes)
L0 I$: 64 instructions (1KByte)
LM Bank: 8KB (32KB total)

Data Path





The Locality Challenge
Data Movement Energy 77pJ/F



Energy Shopping List

Processor Technology	40 nm	10nm
Vdd (nominal)	0.9 V	0.7 V
DFMA energy	50 pJ	7.6 pJ
64b 8 KB SRAM Rd	14 pJ	2.1 pJ
Wire energy (256 bits, 10mm)	310 pJ	174 pJ

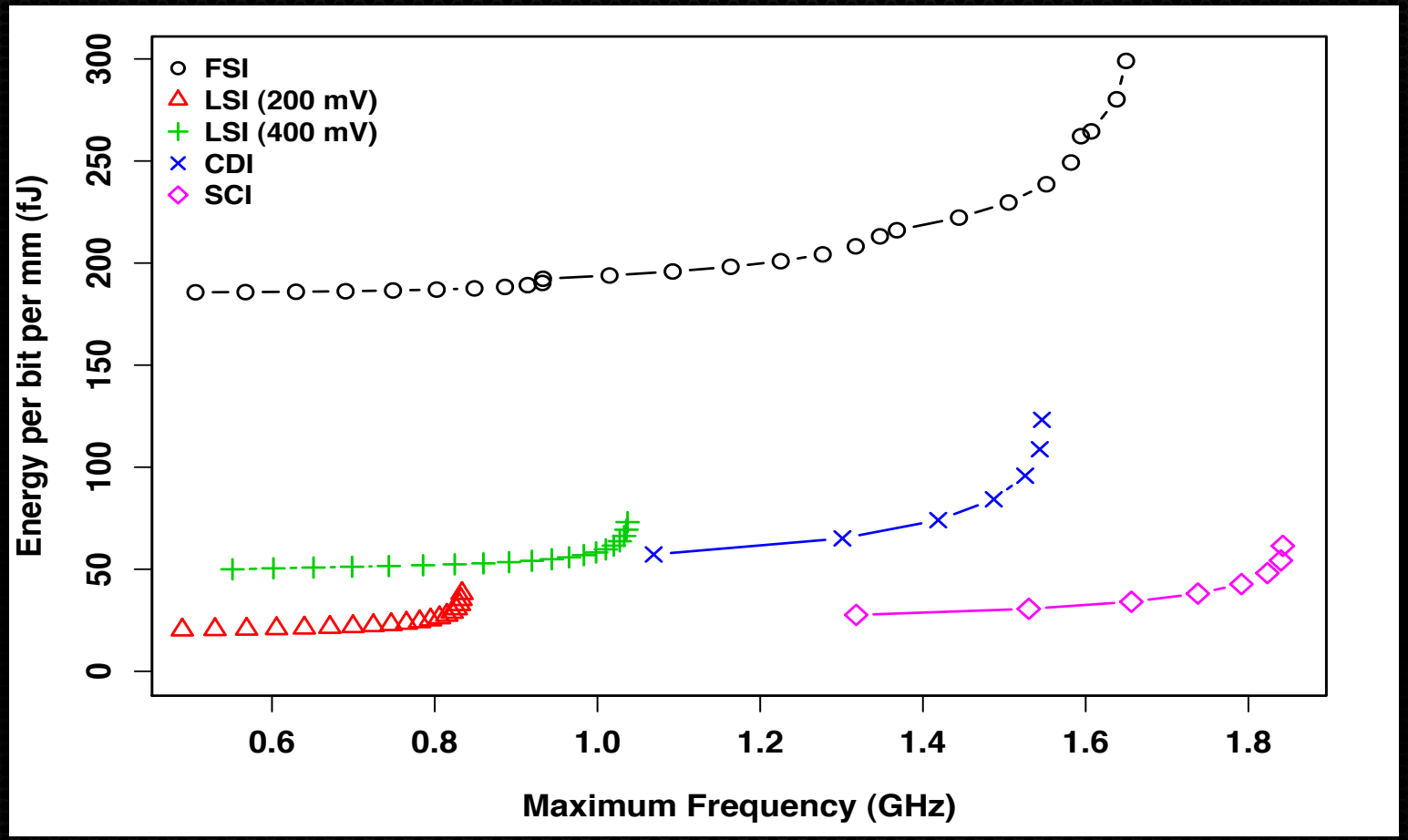
Memory Technology	45 nm	16nm
DRAM interface pin bandwidth	4 Gbps	50 Gbps
DRAM interface energy	20-30 pJ/bit	2 pJ/bit
DRAM access energy	8-15 pJ/bit	2.5 pJ/bit

FP Op lower bound
=
4 pJ

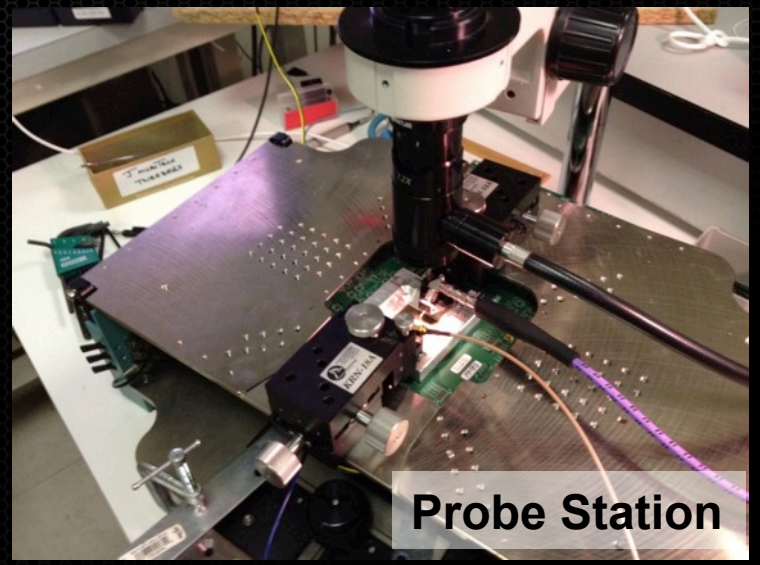
Keckler [Micro 2011], Vogelsang [Micro 2010]

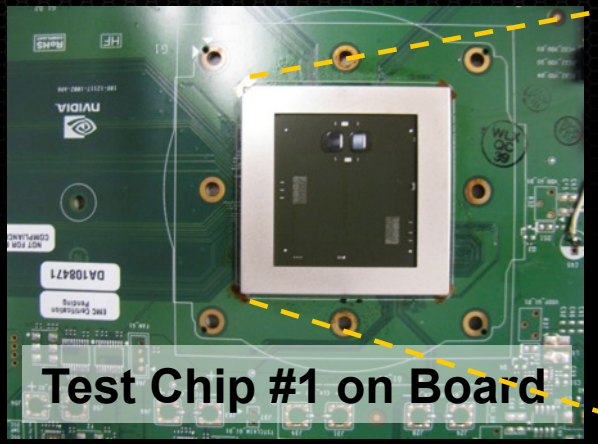
Minimize Data Movement

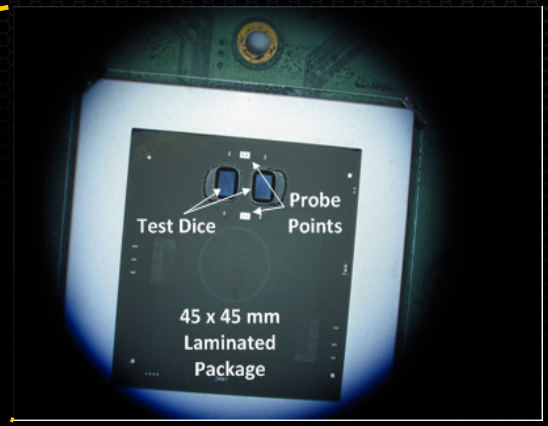
Move Data More Efficiently

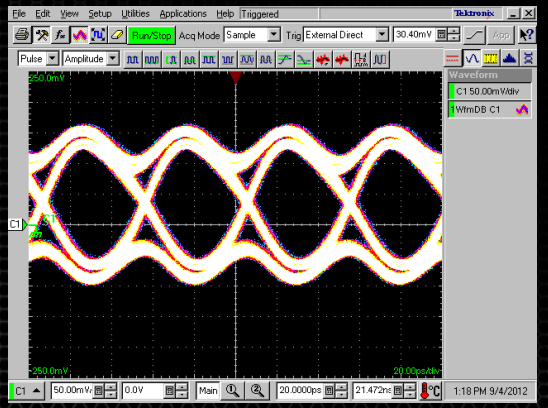

FSI
LSI (200 mV)
LSI (400 mV)
CDI
SCI
Energy per bit per mm (fJ)
0
50
100
150
200
250
300
0.6
0.8
1.0
1.2
1.4
1.6
1.8
Maximum Frequency (GHz)

GRS Test Chips

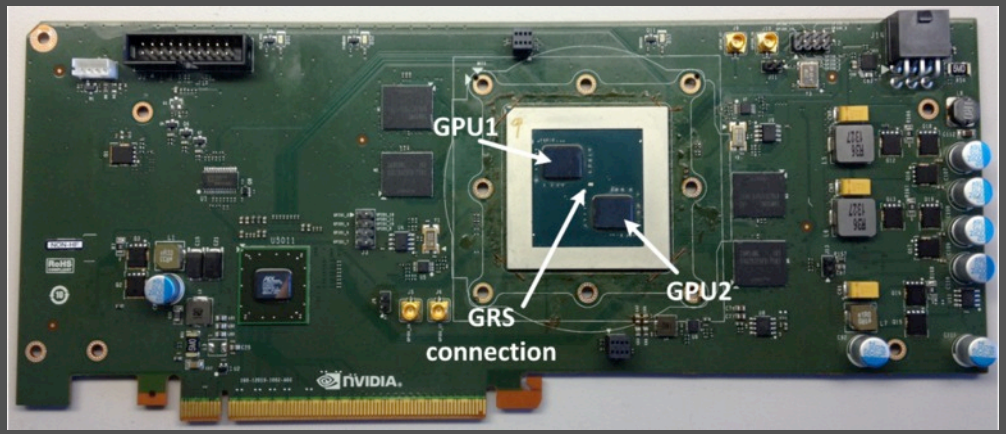








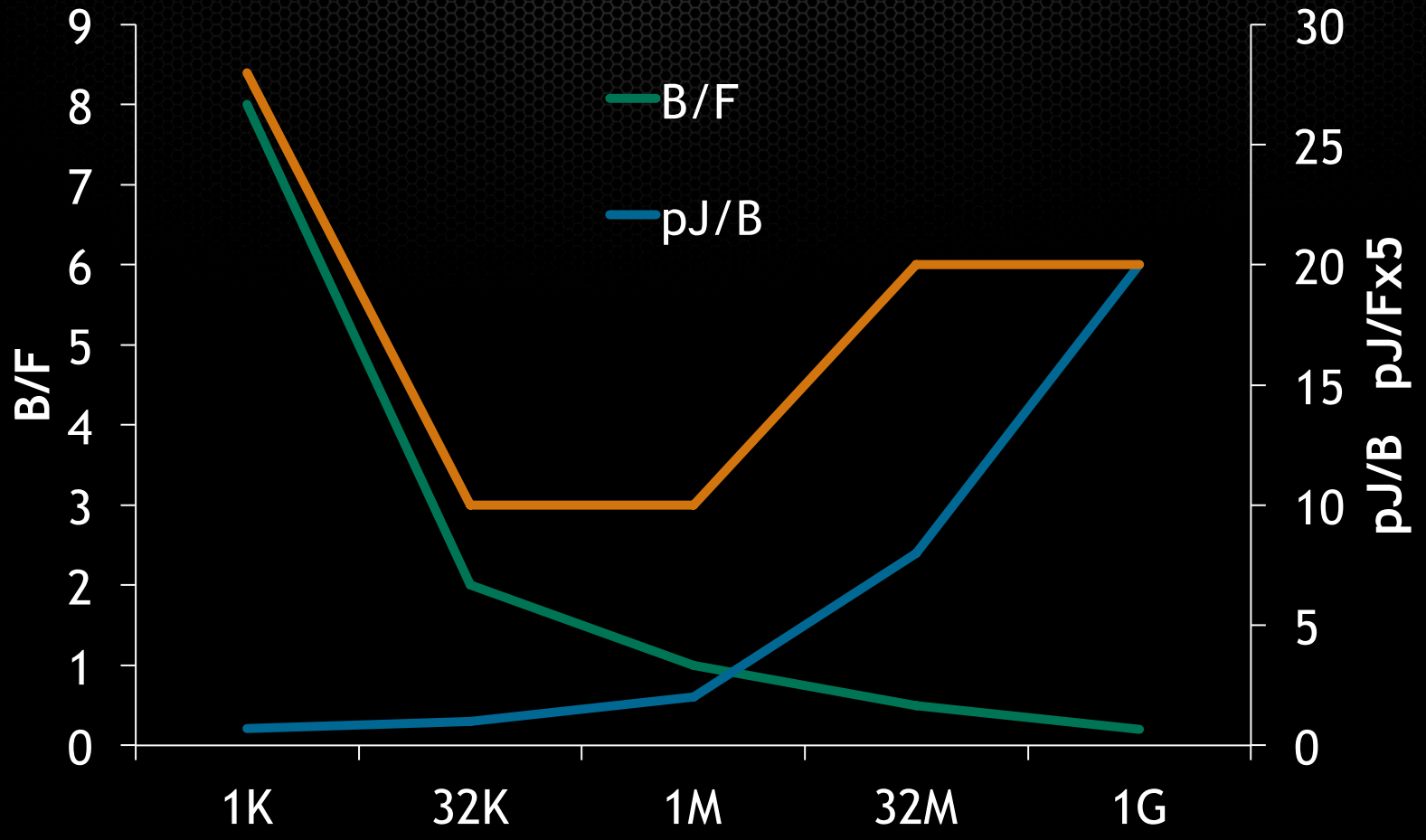
Eye Diagram from Probe



Test Chip #2 fabricated on production GPU

NVIDIA

Optimized Circuits
77pJ/F -> 18pJ/F



NVIDIA

Simplify Programming

While Improving Locality

Parallel programming is not inherently any more difficult than serial programming

However, we can make it a lot more difficult

A simple parallel program

```
forall molecule in set { // launch a thread array
    forall neighbor in molecule.neighbors { // nested
        forall force in forces { // doubly nested
           molecule.force =
             reduce_sum(force(molecule, neighbor))
        }
    }
}
```

Why is this easy?

```
forall molecule in set { // launch a thread array
    forall neighbor in molecule.neighbors { // nested
        forall force in forces { // doubly nested
           molecule.force =
             reduce_sum(force(molecule, neighbor))
        }
    }
}
```

No machine details
All parallelism is expressed
Synchronization is semantic (in reduction)

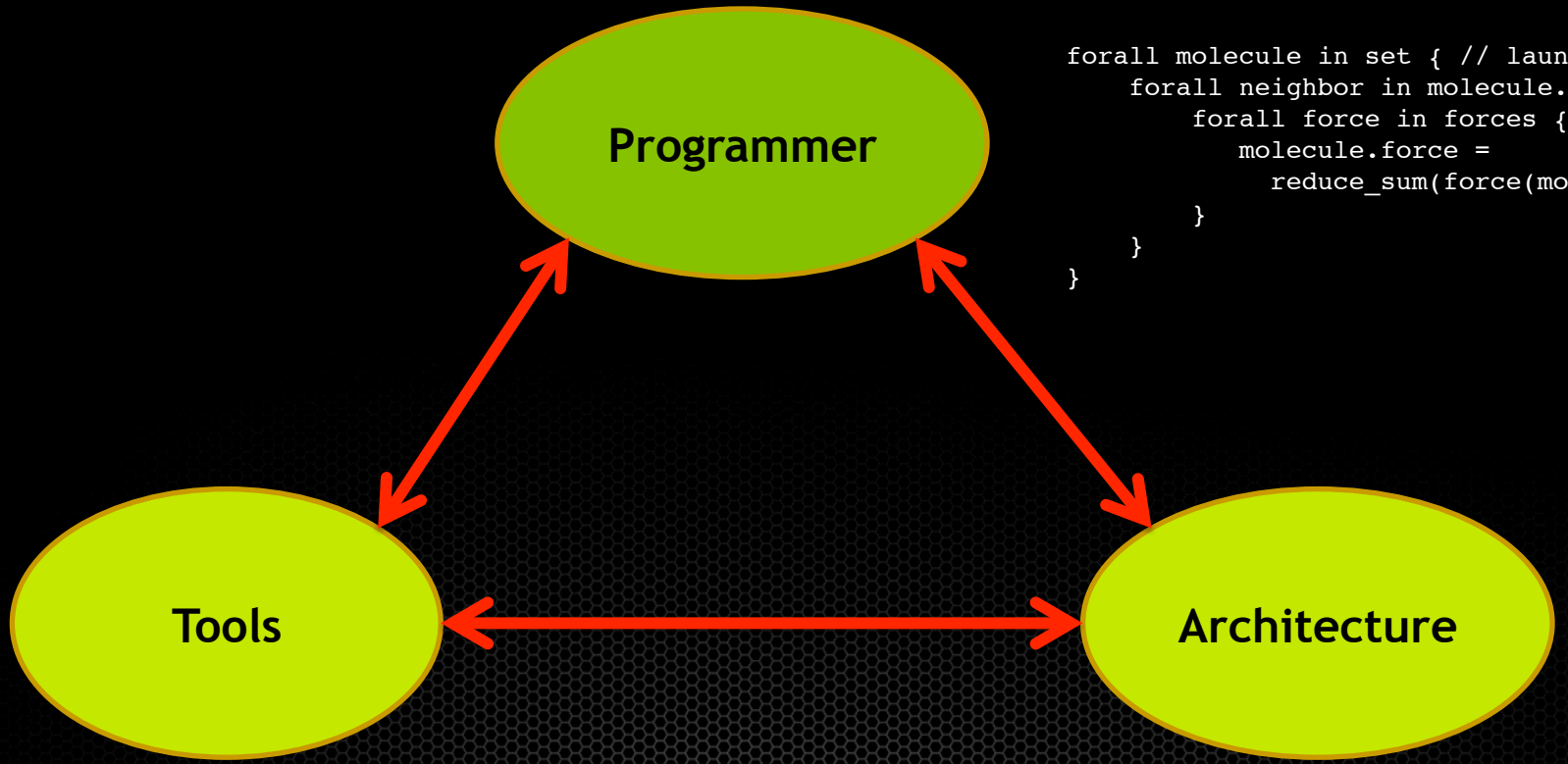
```
pid = fork() ; // explicitly managing threads

lock(struct.lock) ;  // complicated, error-prone synchronization
// manipulate struct
unlock(struct.lock) ;

code = send(pid, tag, &msg) ;  // partition across nodes
```

We could make it hard

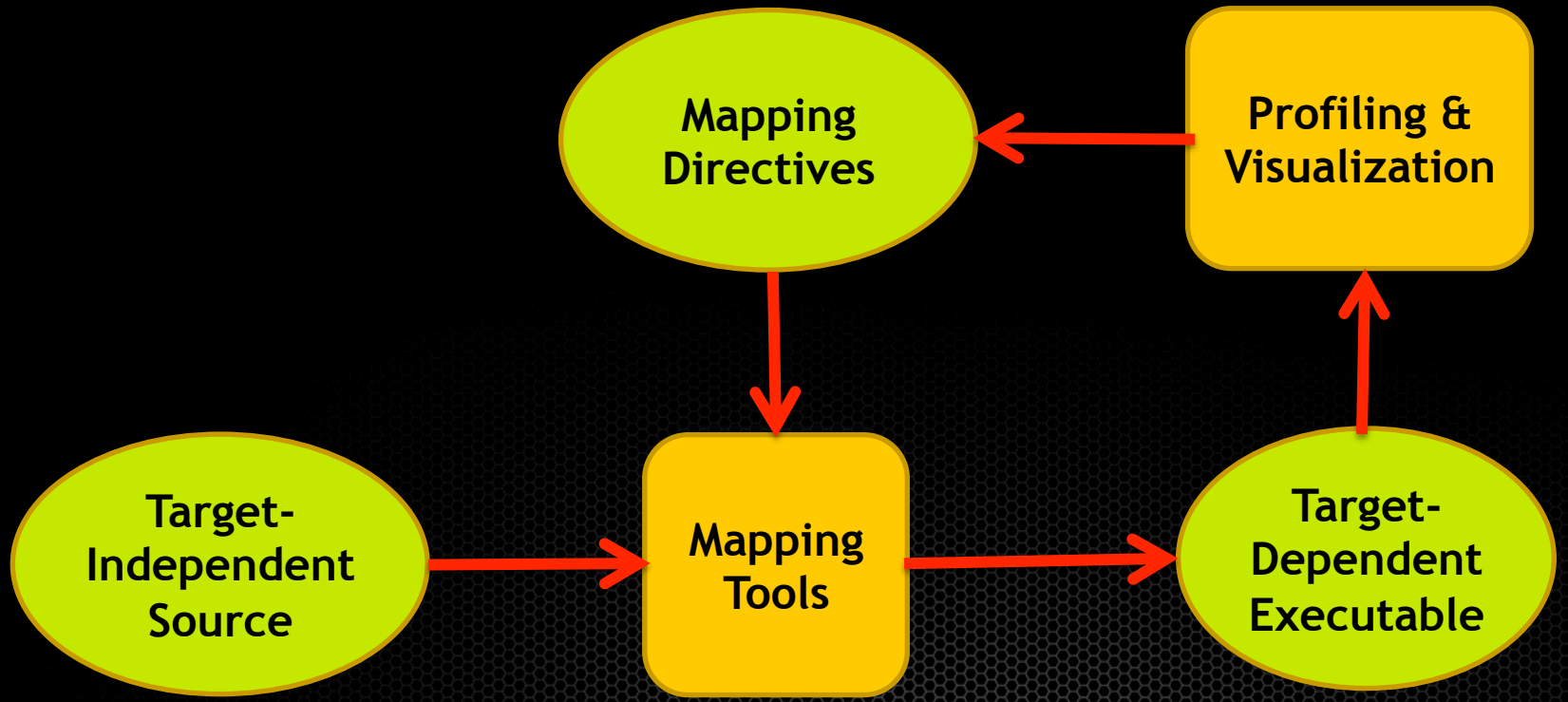
Programmers, tools, and architecture Need to play their positions



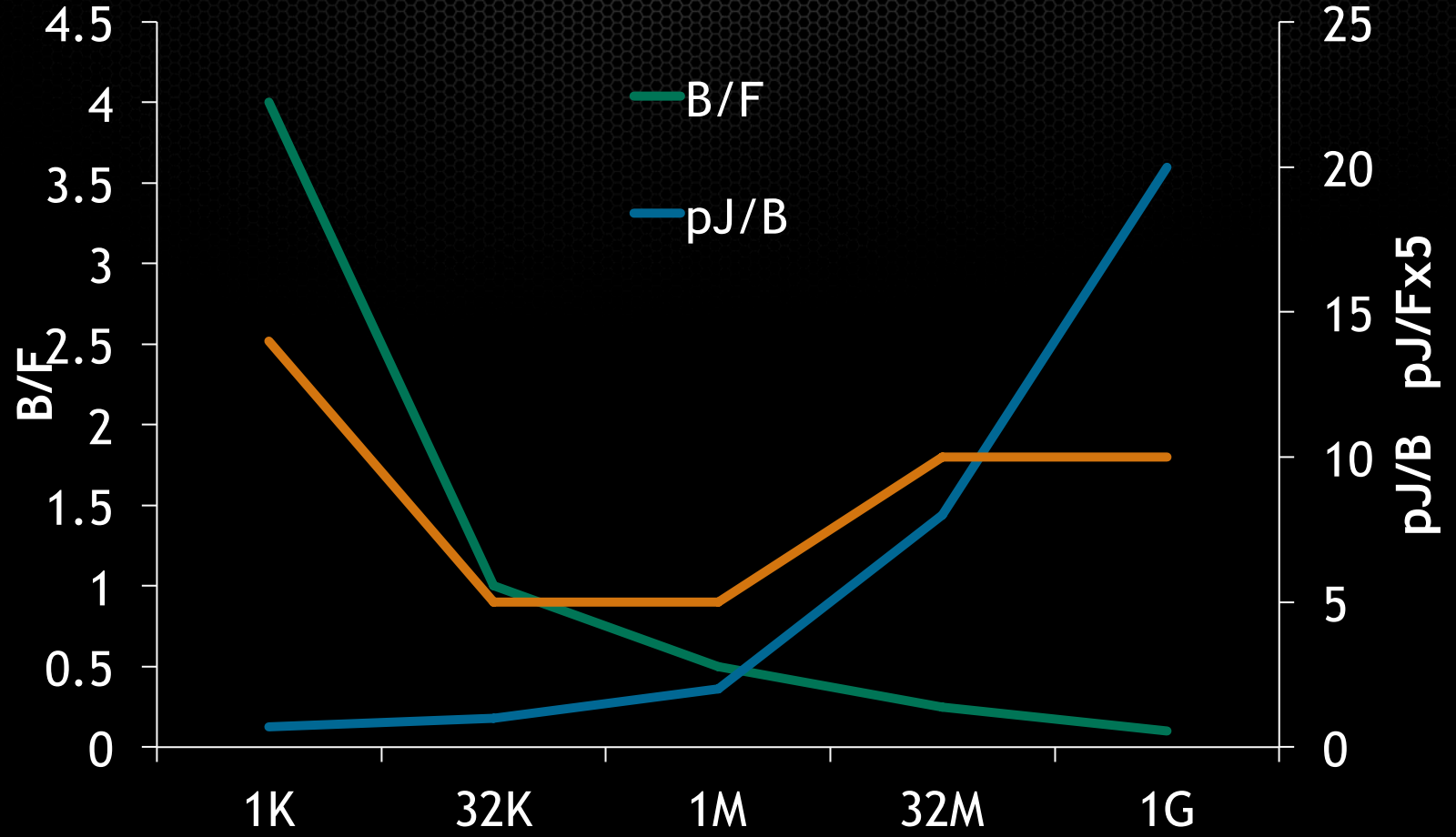

```
forall molecule in set { // launch a thread array
    forall neighbor in molecule.neighbors { //
        forall force in forces { // doubly nested
            molecule.force =
                reduce_sum(force(molecule, neighbor))
        }
    }
}
```

Map foralls in time and space
Map molecules across memories
Stage data up/down hierarchy
Select mechanisms

Exposed storage hierarchy
Fast comm/sync/thread mechanisms


Mapping Directives
Profiling & Visualization
Target-Independent Source
Mapping Tools
Target-Dependent Executable

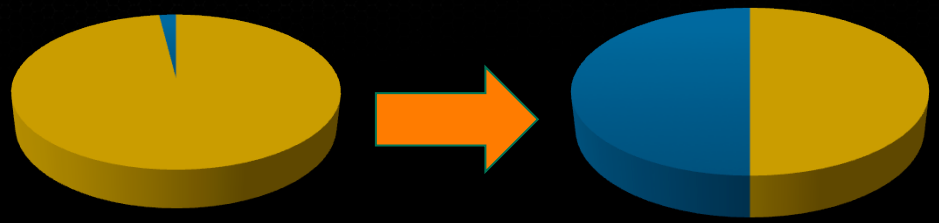
Autotuned Software
18pJ/F -> 9pJ/F



NVIDIA

Conclusion

- Power-limited: from data centers to cell phones
  - Perf/W is Perf
- Throughput cores
  - Reduce overhead



- Data movement
  - Circuits: 200 -> 20
  - Optimized software
- Parallel programming is simple – we can make it hard
  - Target-independent programming – mapping via tools


“Super” Computing
From Super Computers to Super Phones