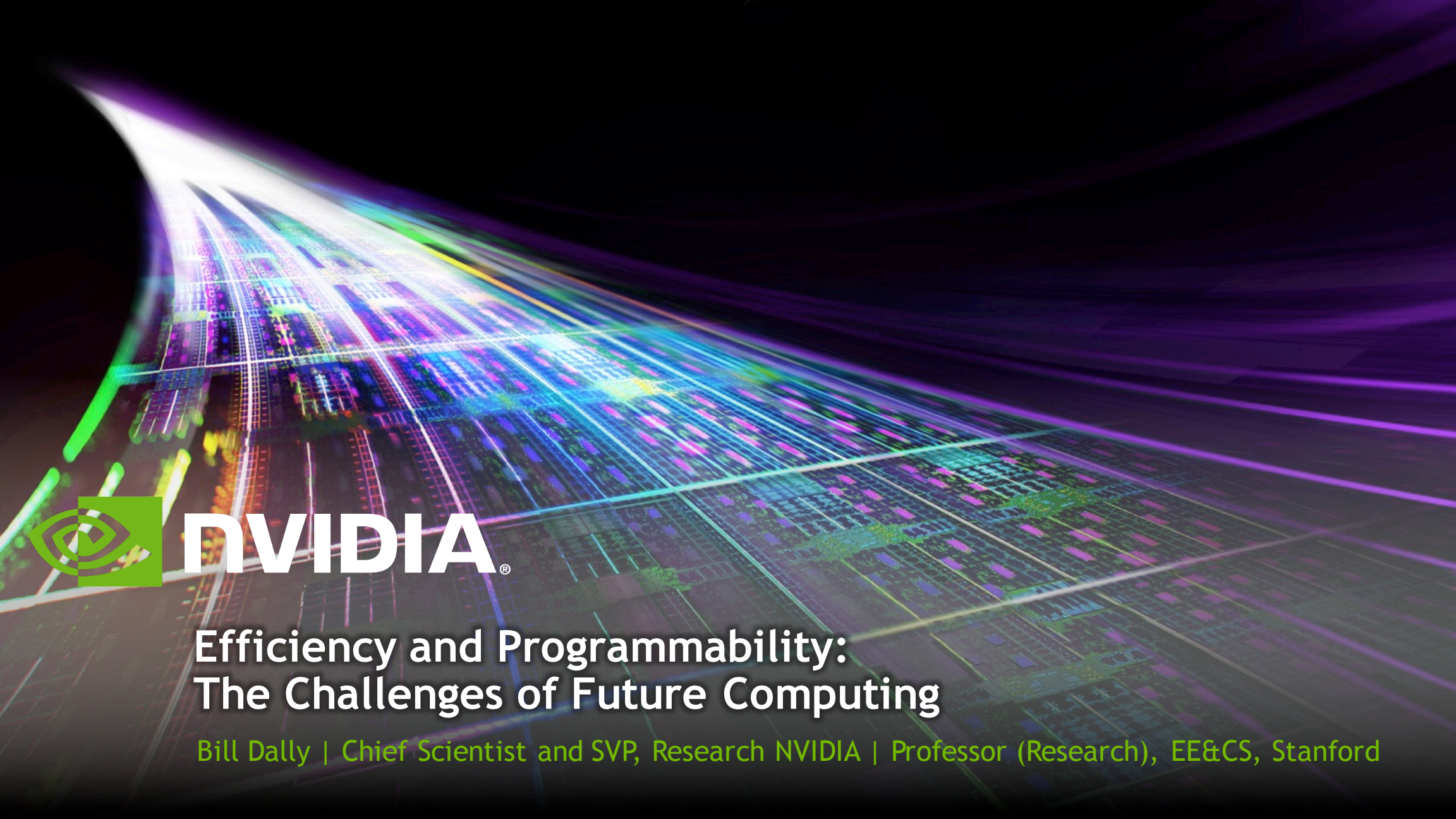
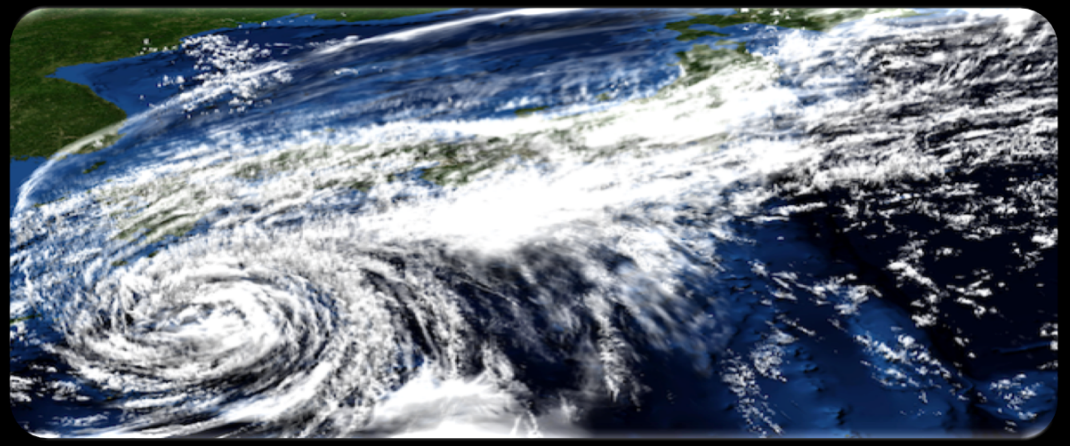
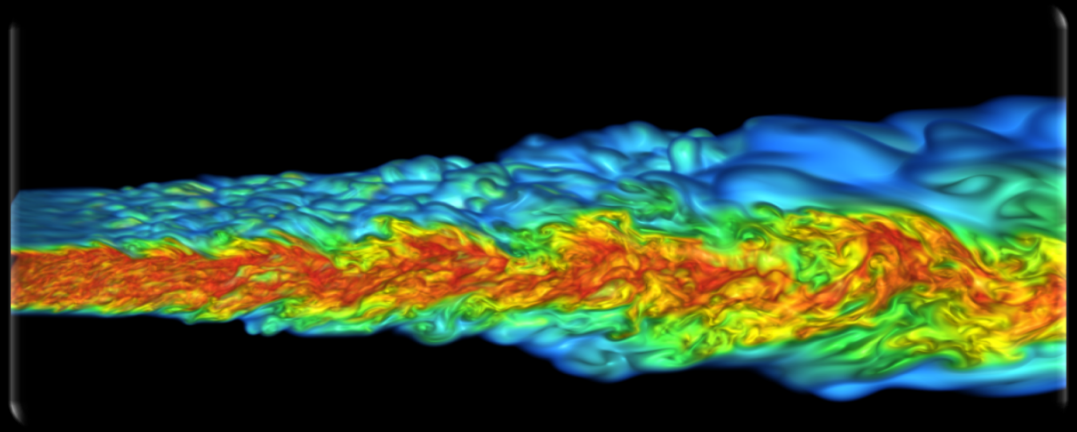


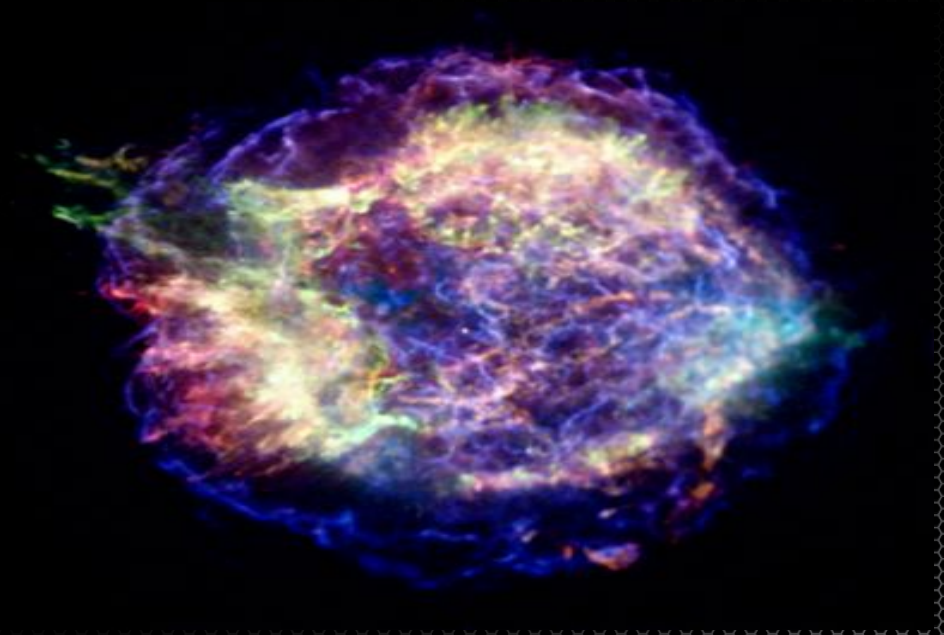
Efficiency and Programmability: The Challenges of Future Computing

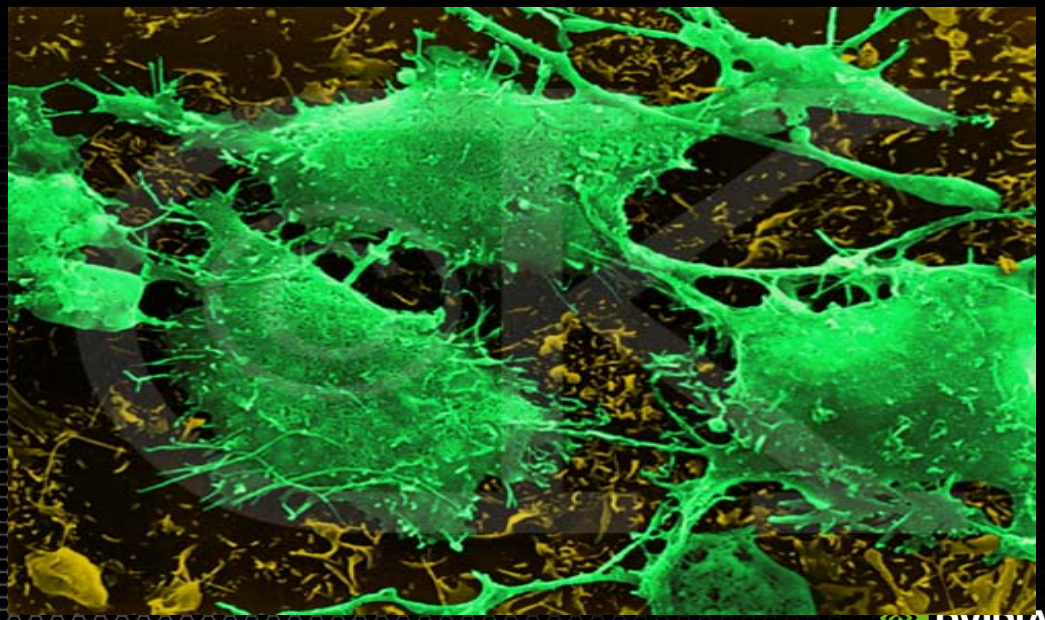
Bill Dally | Chief Scientist and SVP, Research NVIDIA | Professor (Research), EE&CS, Stanford

Computing Is Enabling Transformational Science









Computing Is Enabling Transformational Science

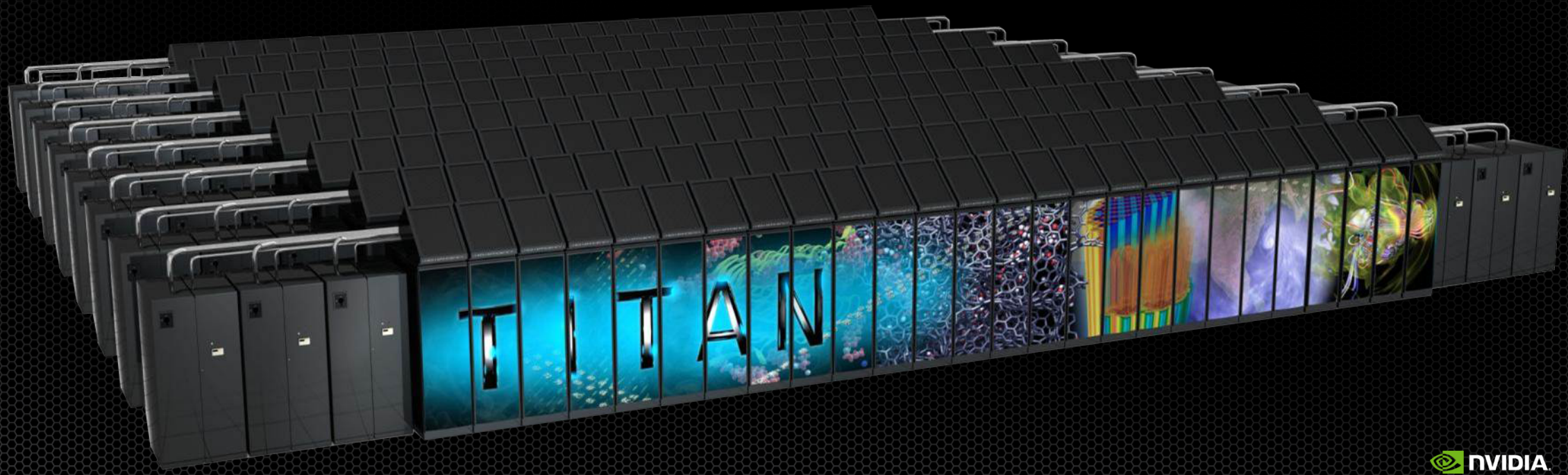
High-Performance Computers are Scientific Instruments

TITAN

18,688 NVIDIA Tesla K20X GPUs

27 Petaflops Peak: 90% of Performance from GPUs

17.59 Petaflops Sustained Performance on Linpack



NVIDIA

US to Build Two Flagship Supercomputers

OAK RIDGE National Laboratory

Lawrence Livermore National Laboratory

SUMMIT

SIERRA

150-300 PFLOPS Peak Performance

IBM POWER9 CPU + NVIDIA Volta GPU

NVLink High Speed Interconnect

40 TFLOPS per Node, >3,400 Nodes

2017

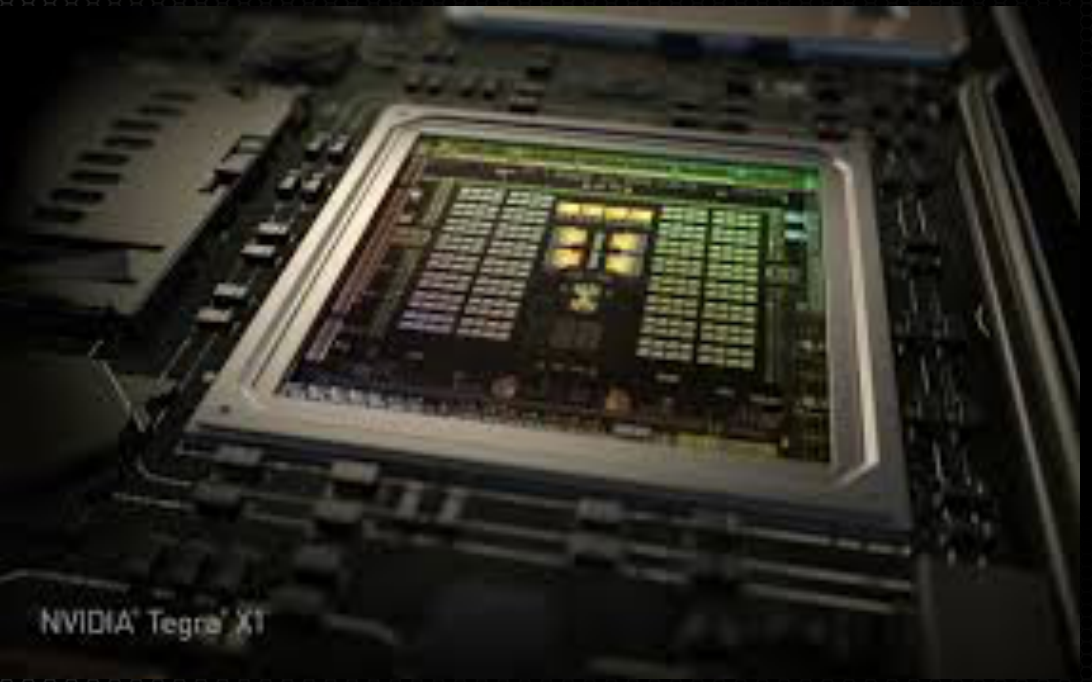
Major Step Forward on the Path to Exascale

NVIDIA


htc
10:08 AM
10
08
AM
Sun, Feb 26
Barcelona
Mostly Sunny
12°C
16°C / 8°C
Play Store
Favorites

NVIDIA

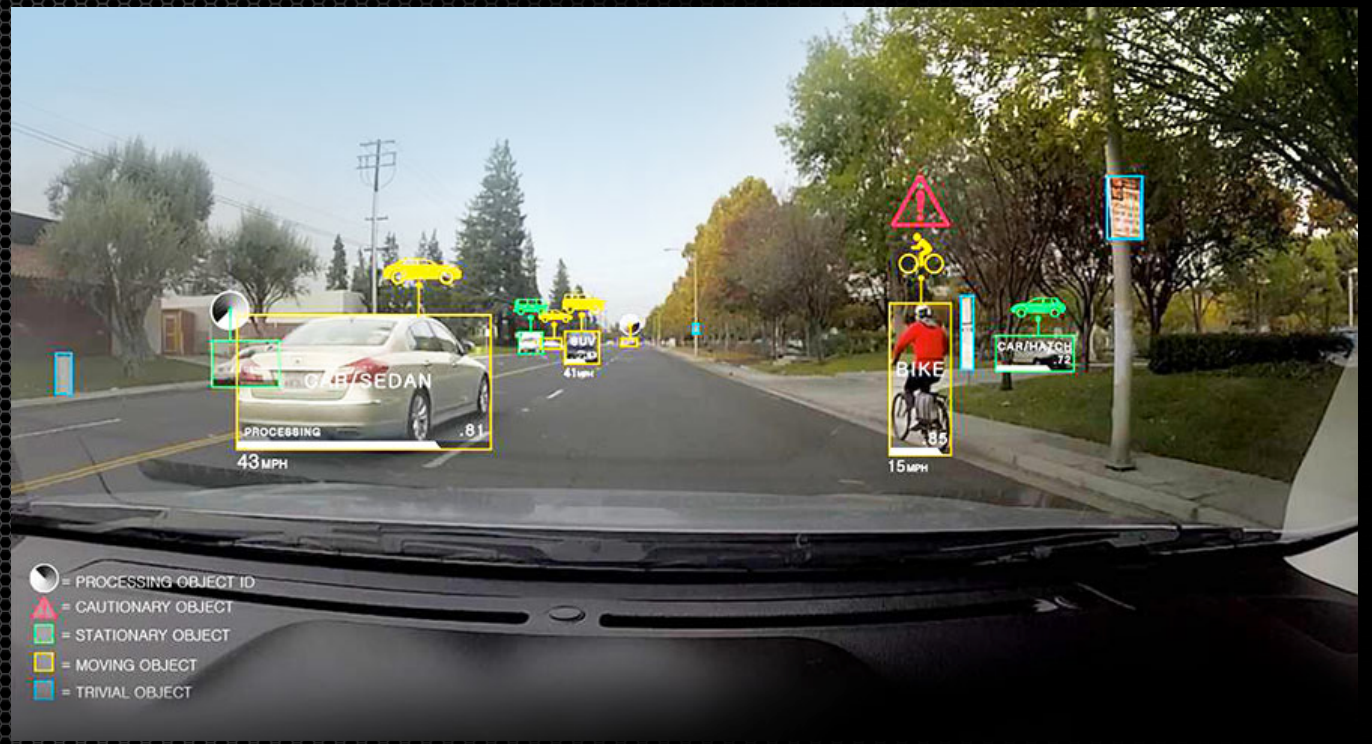
Tegra X1

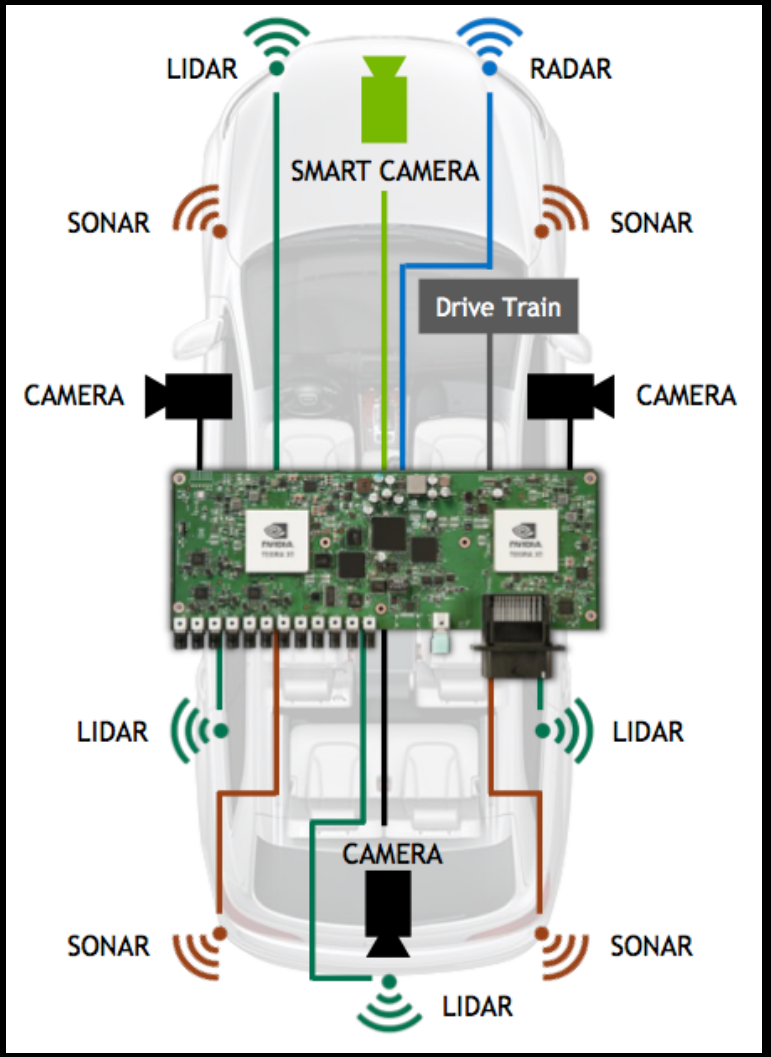


- 256 CUDA cores @ ~1 GHz
- 1 Teraflop FP16
- 4GB of LPDDR4 @ 25.6 GB/s
- 15 W TDP (1W idle, <10W typical)
- 100GFLOPS/W (FP16)
- 20nm process

NVIDIA

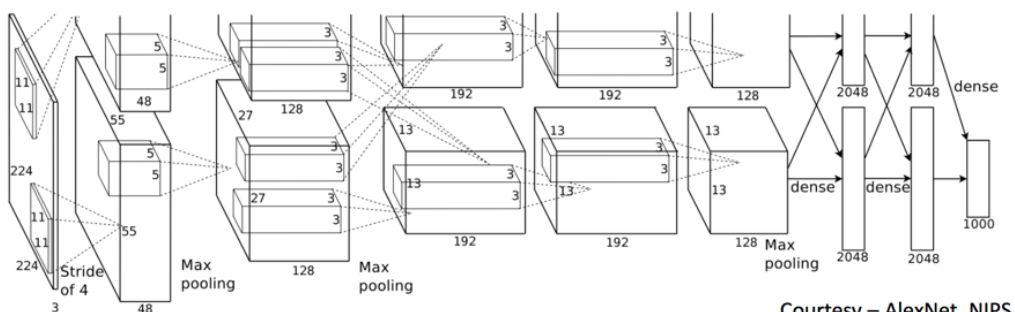
Making Transportation Safer





NVIDIA

Hardware and Data enable DNNs



Courtesy – AlexNet, NIPS 2012

2 Trends

The End of Dennard Scaling

Pervasive Parallelism

2 Trends

The End of Dennard Scaling

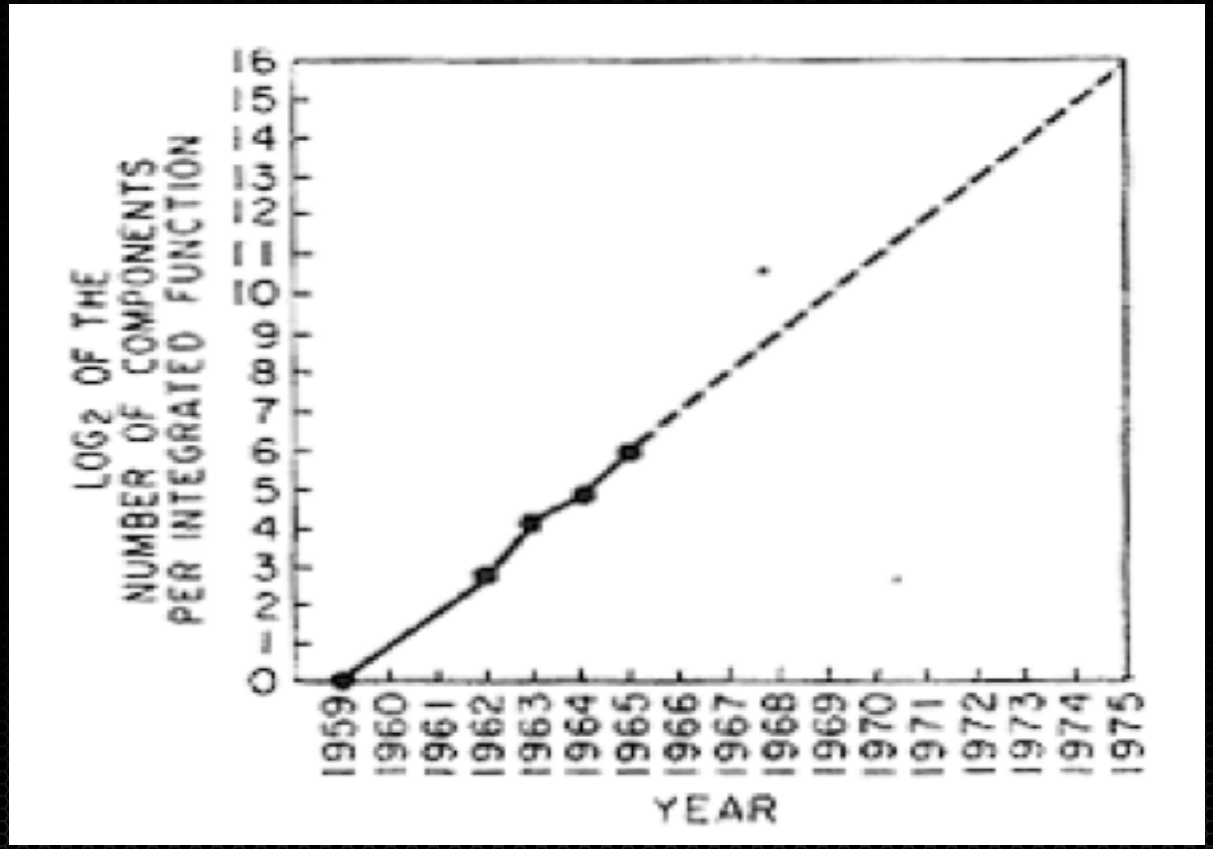
Pervasive Parallelism

Moore’s Law doubles processor performance every few years, right?

Moore's Law doubles processor performance every few years, right?

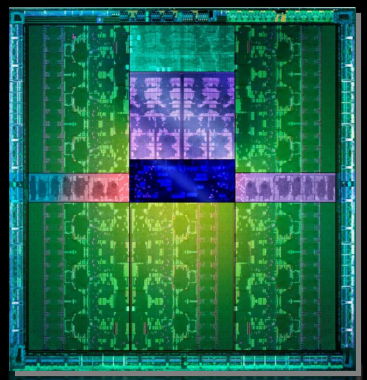
Wrong!

Moore's Law gives us transistors Which we used to turn into scalar performance

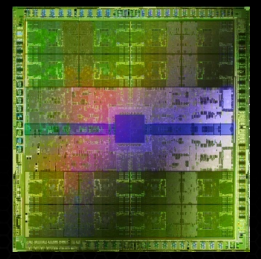


Moore, Electronics 38(8) April 19, 1965

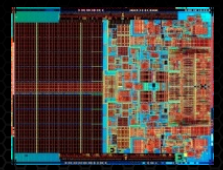
NVIDIA



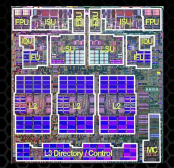
2012: 7B Tx



2010: 3B Tx



2007: 580M Tx



2004: 275M Tx

2001: 42M Tx

1997: 7.5M Tx

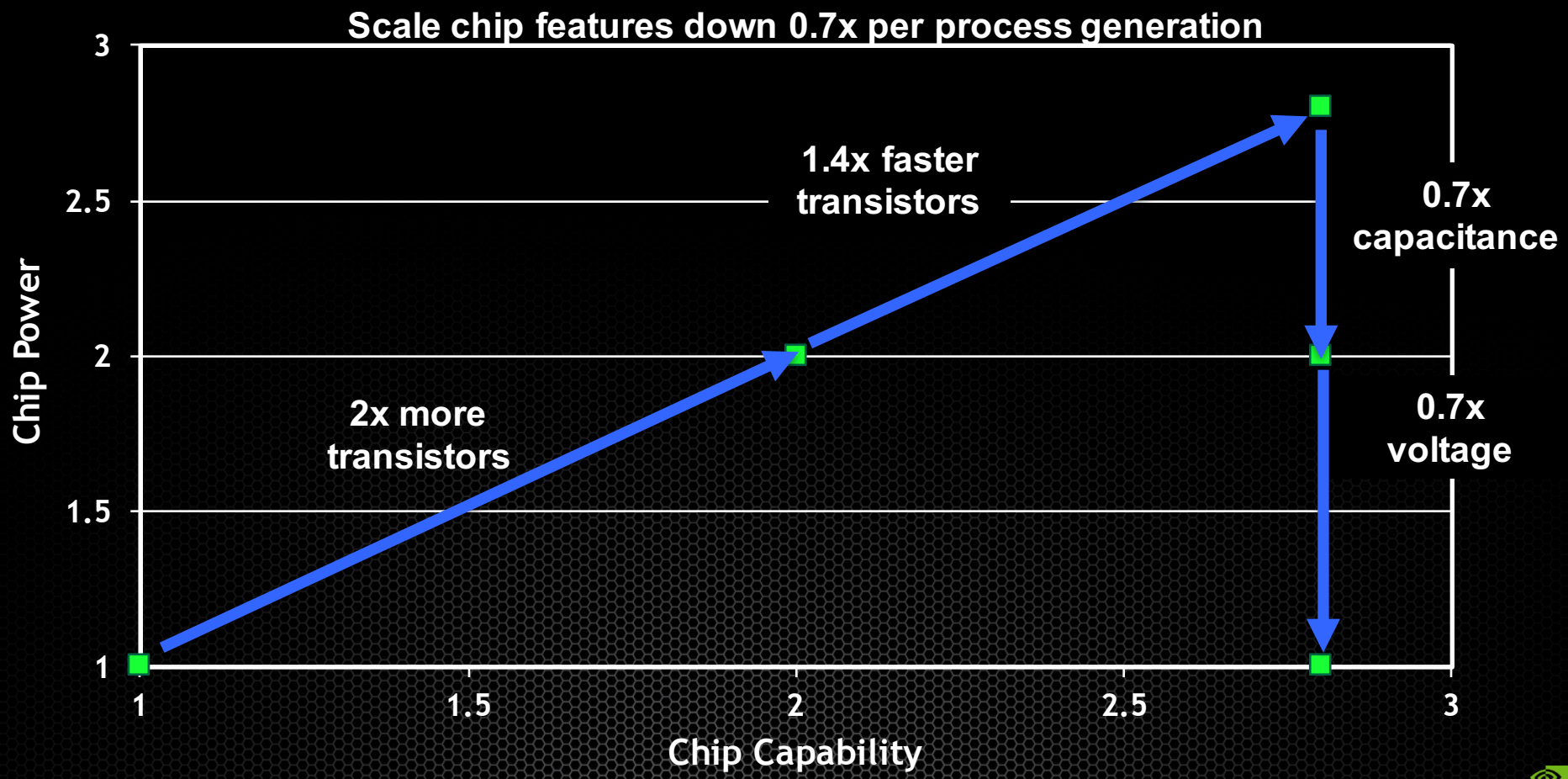
1993: 3M Tx

Moore's Law is Only Part of the Story

NVIDIA

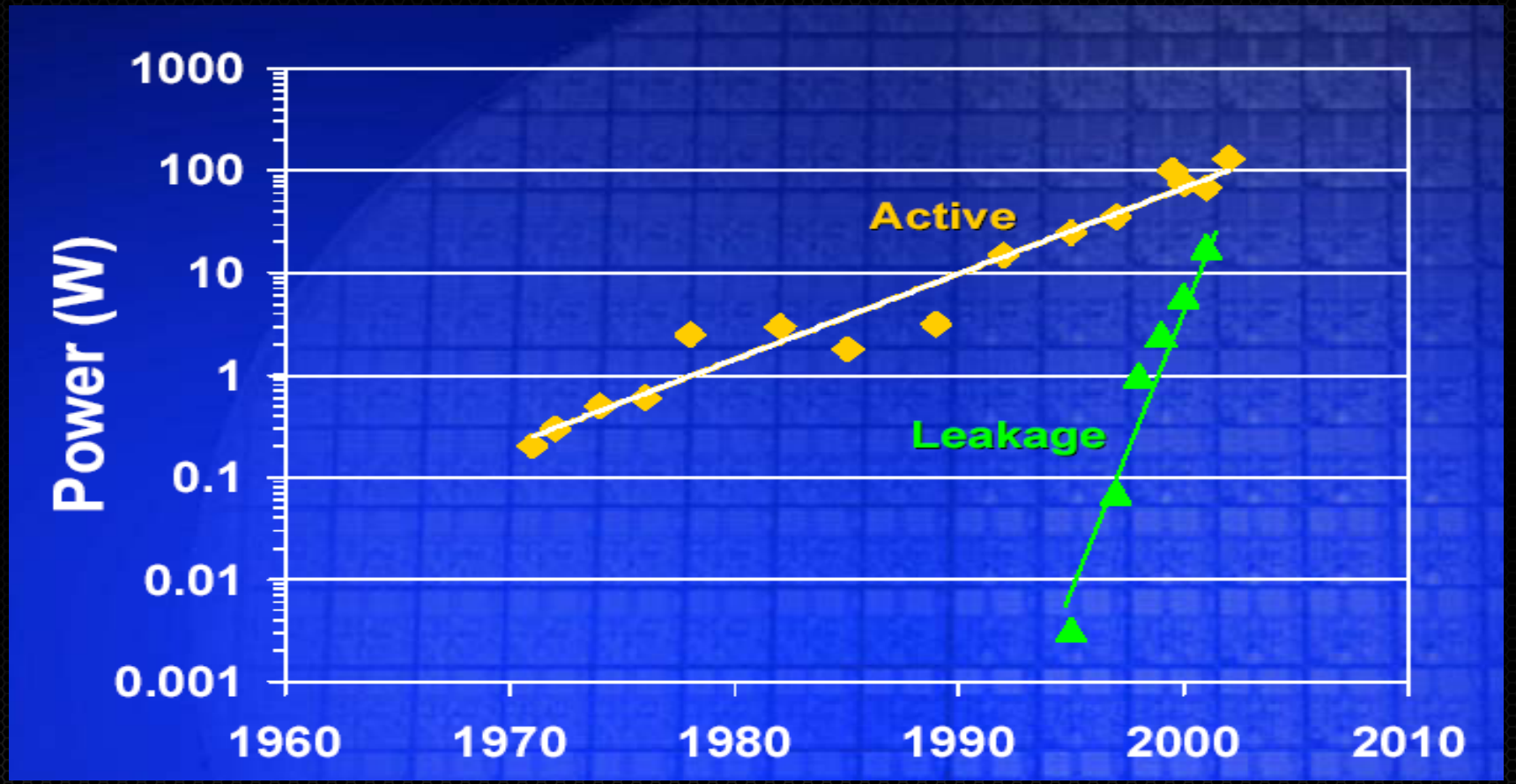
Classic Dennard Scaling

2.8x chip capability in same power



NVIDIA

But L^3 energy scaling ended in 2005

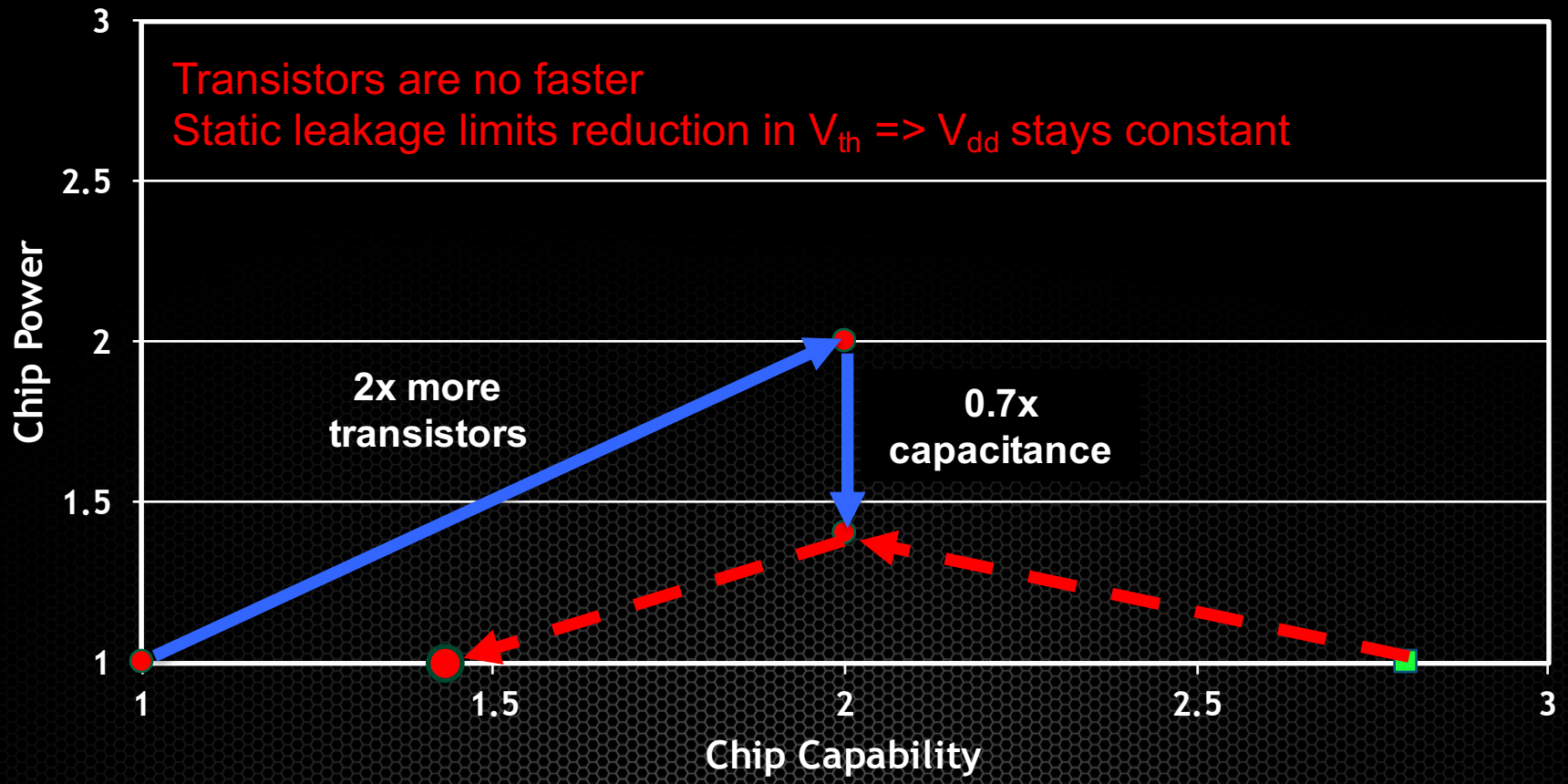


Moore, ISSCC Keynote, 2003

NVIDIA

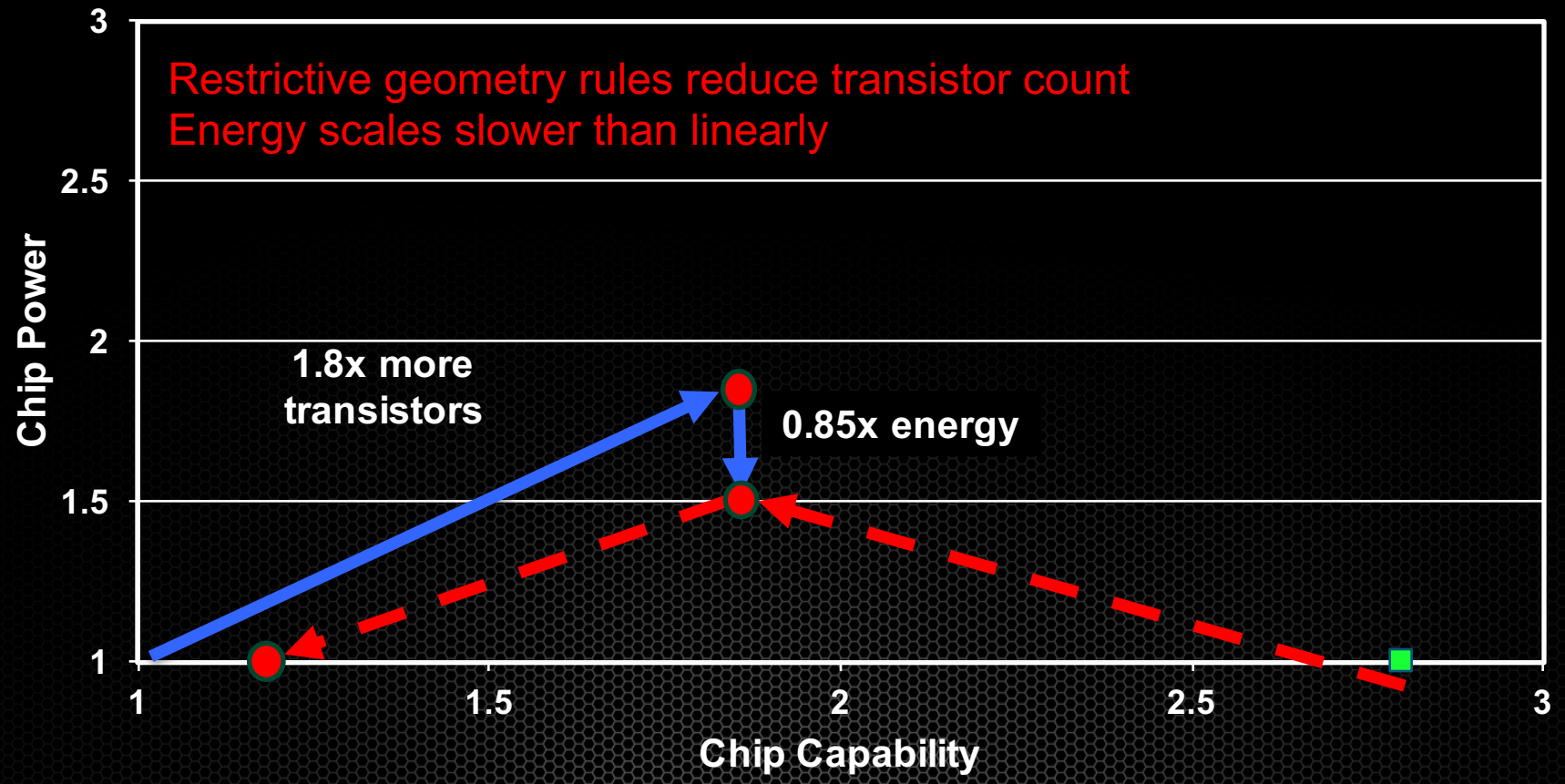
Post Dennard Scaling

2x chip capability at 1.4x power
1.4x chip capability at same power



Reality isn't even this good

1.8x chip capability at 1.5x power
1.2x chip capability at same power

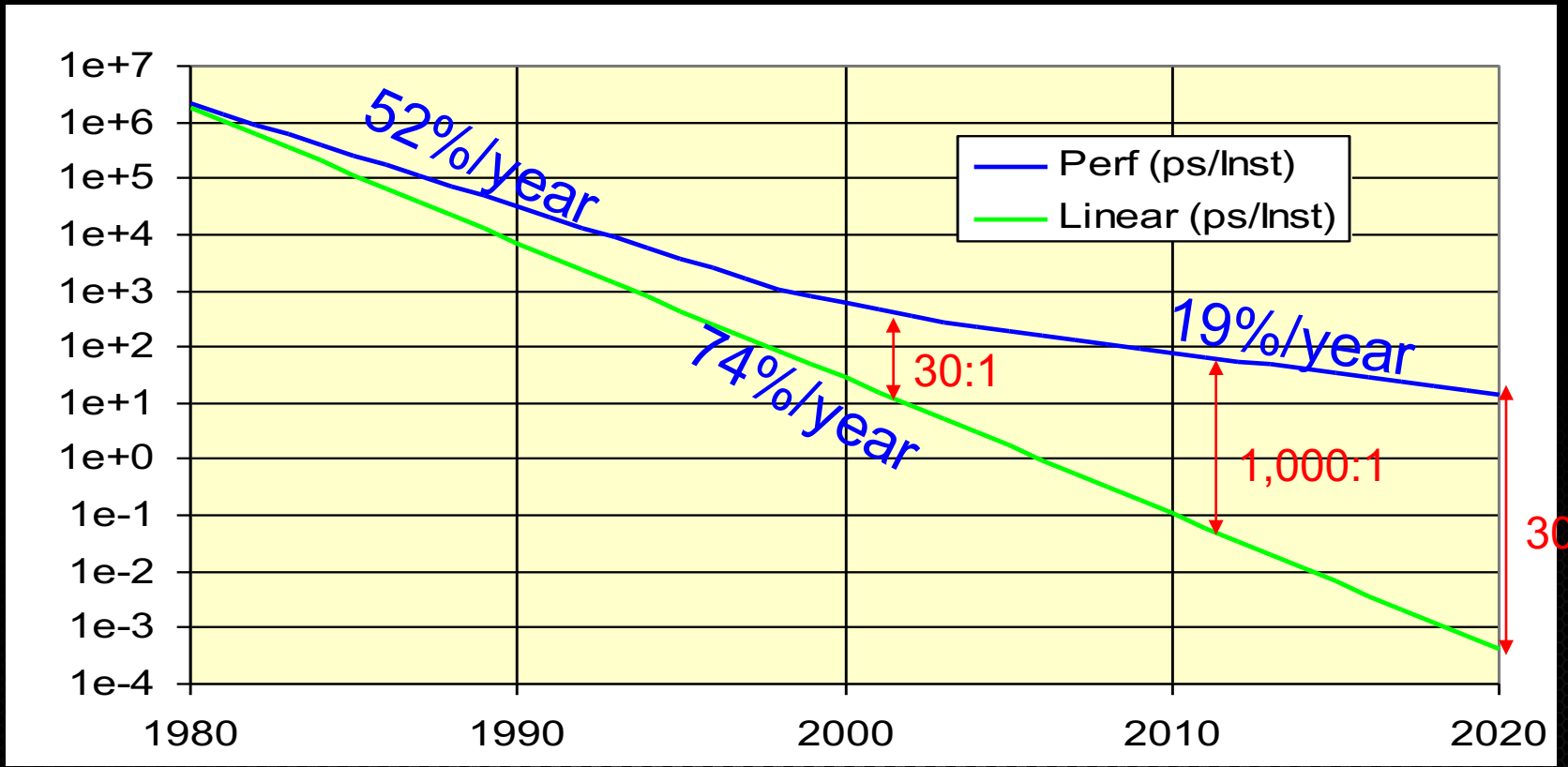


NVIDIA

"Moore's Law gives us more transistors… Dennard scaling made them useful."

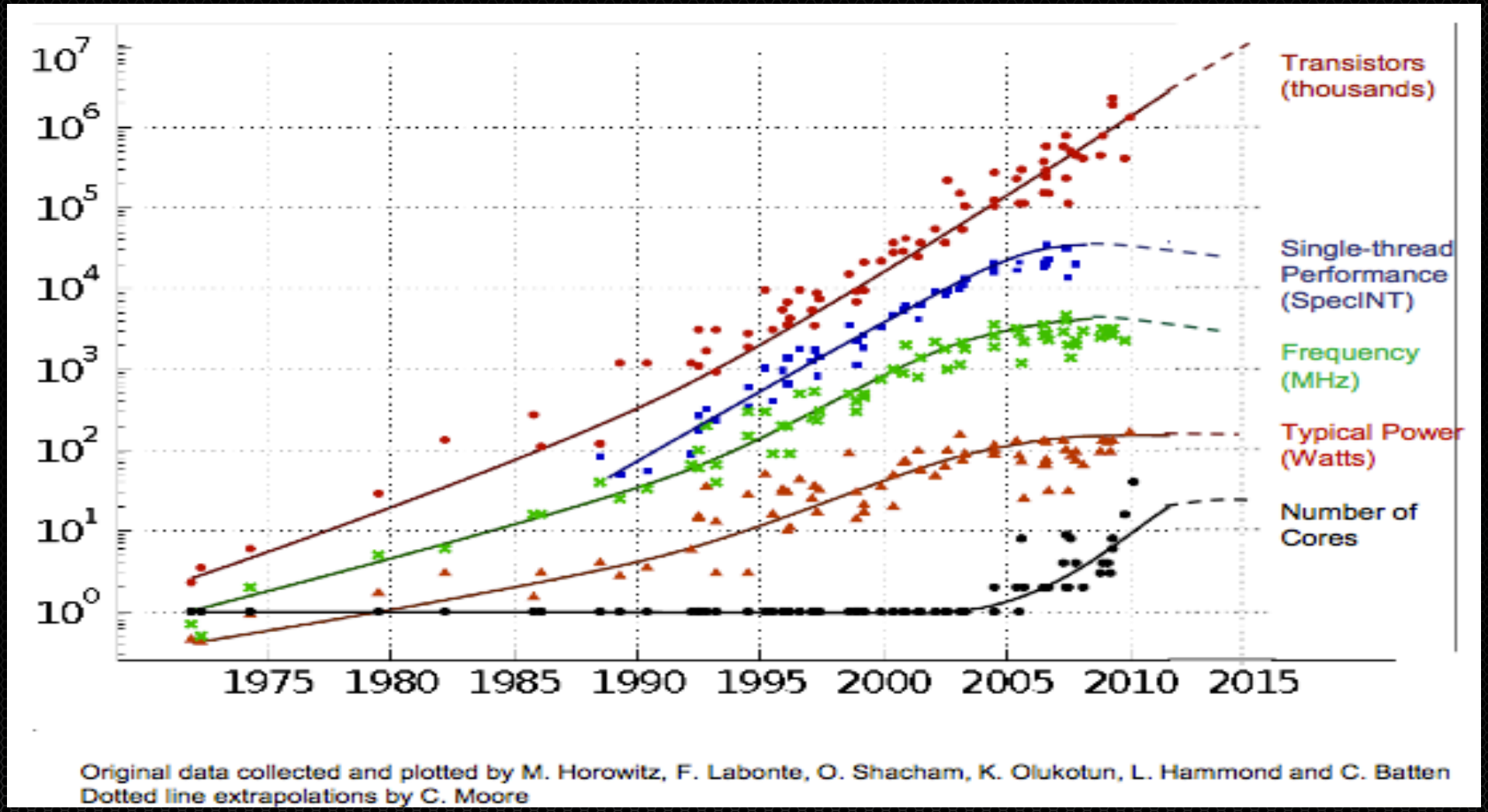
Bob Colwell, DAC 2013, June 4, 2013

Also, ILP was 'mined out' in 2000



Dally et al. "The Last Classical Computer", ISAT Study, 2001

Result: The End of Historic Scaling



C Moore, *Data Processing in ExaScale-ClassComputer Systems*, Salishan, April 2011

The End of Dennard Scaling

- Processors aren't getting faster, just wider
  - Future gains in performance are from parallelism
- Future systems are energy limited
  - Efficiency ***IS*** Performance
- Process matters less
  - One generation is 1.2x, not 2.8x

Its not about the FLOPs

- DFMA 0.01mm^2 10pJ/OP – 2GFLOPs

16nm chip, 10mm on a side, 200W

A chip with 10^4 FPUs:

100mm^2

200W

20TFLOPS

Pack 50,000 of these in racks

1EFLOPS

10MW

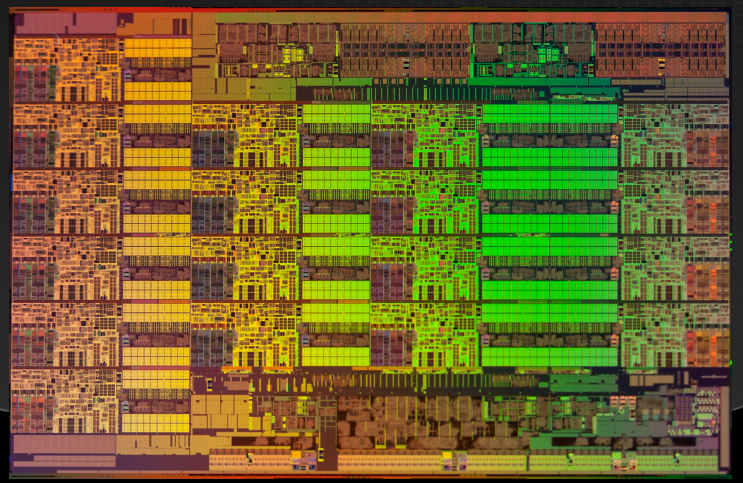
Overhead

Locality

CPU
130 pJ/flop (SP)

Optimized for Latency

Deep Cache Hierarchy

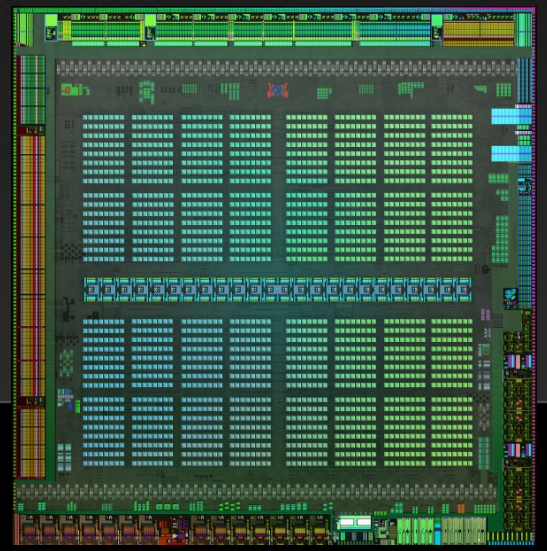


Haswell
22 nm

GPU
30 pJ/flop (SP)

Optimized for Throughput

Explicit Management
of On-chip Memory



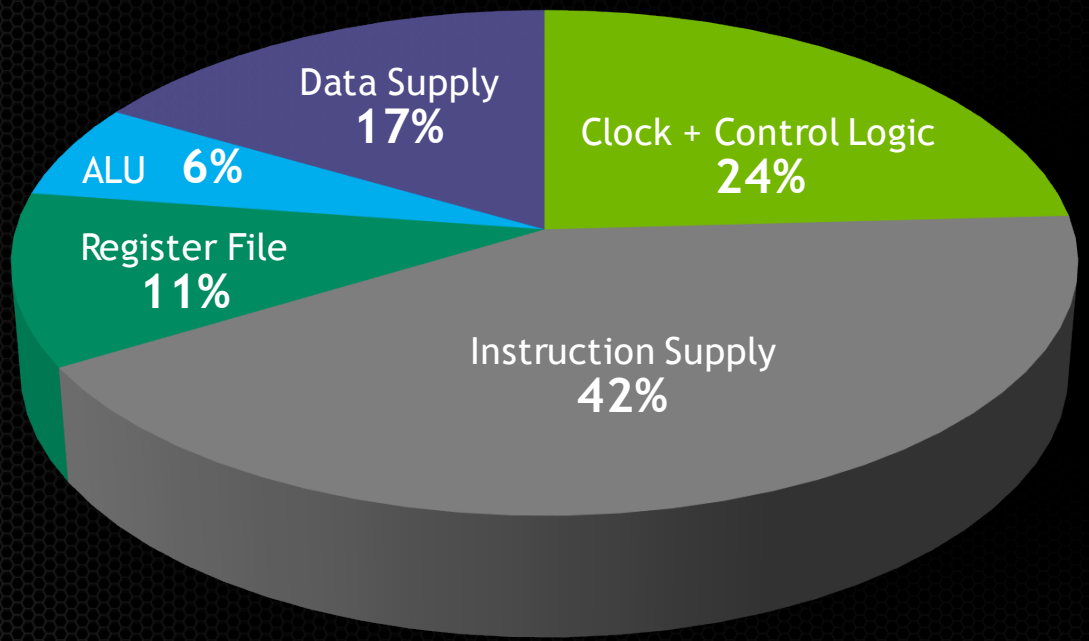
Maxwell
28 nm

Fixed-Function Logic is Even More Efficient

	Energy/Op
CPU (scalar)	1.7nJ
GPU	30pJ
Fixed-Function	3pJ

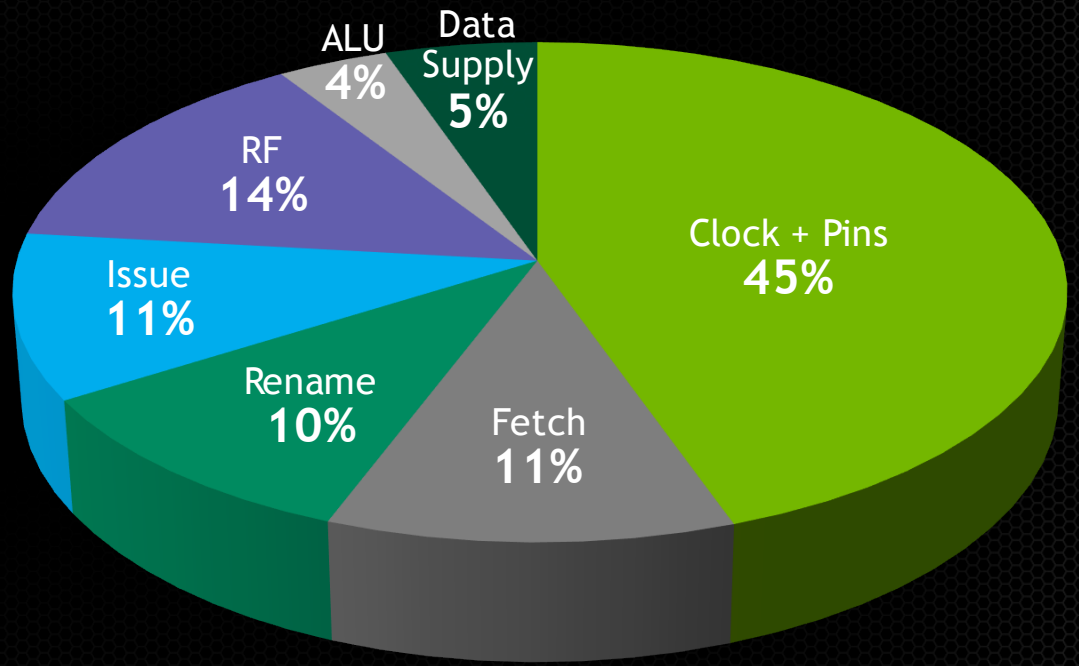
How is Power Spent in a CPU?

In-order Embedded

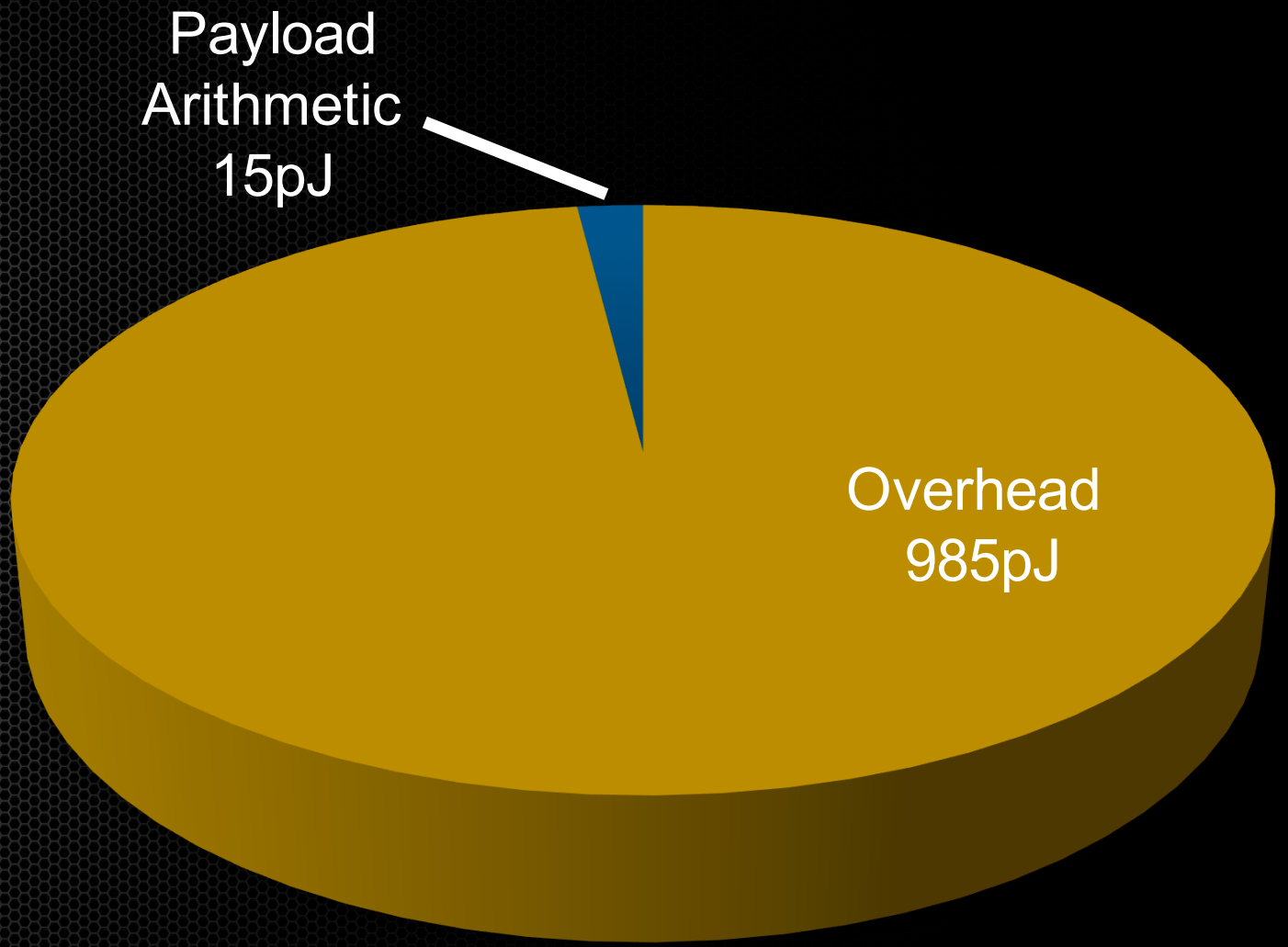


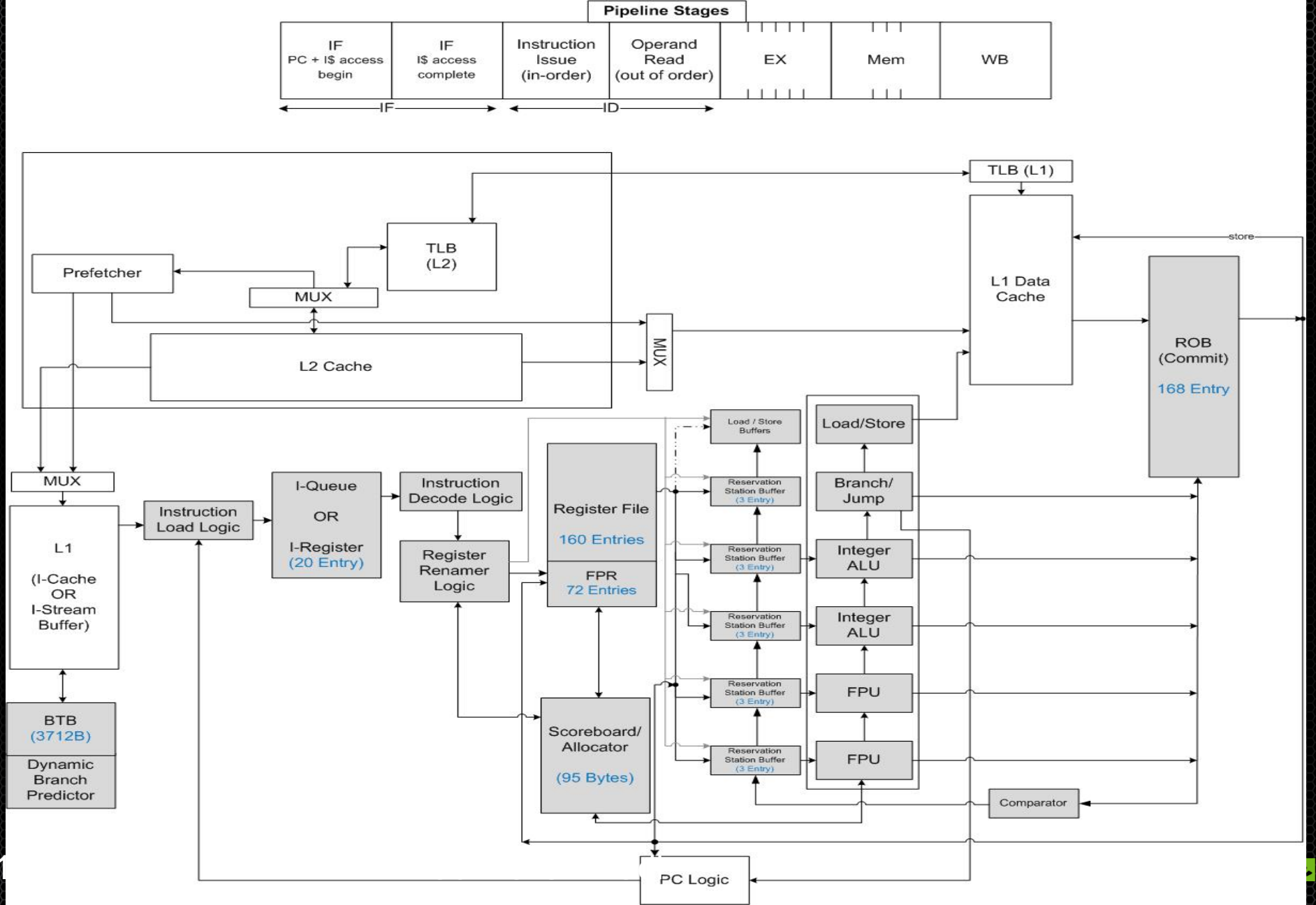
Dally [2008] (Embedded in-order CPU)

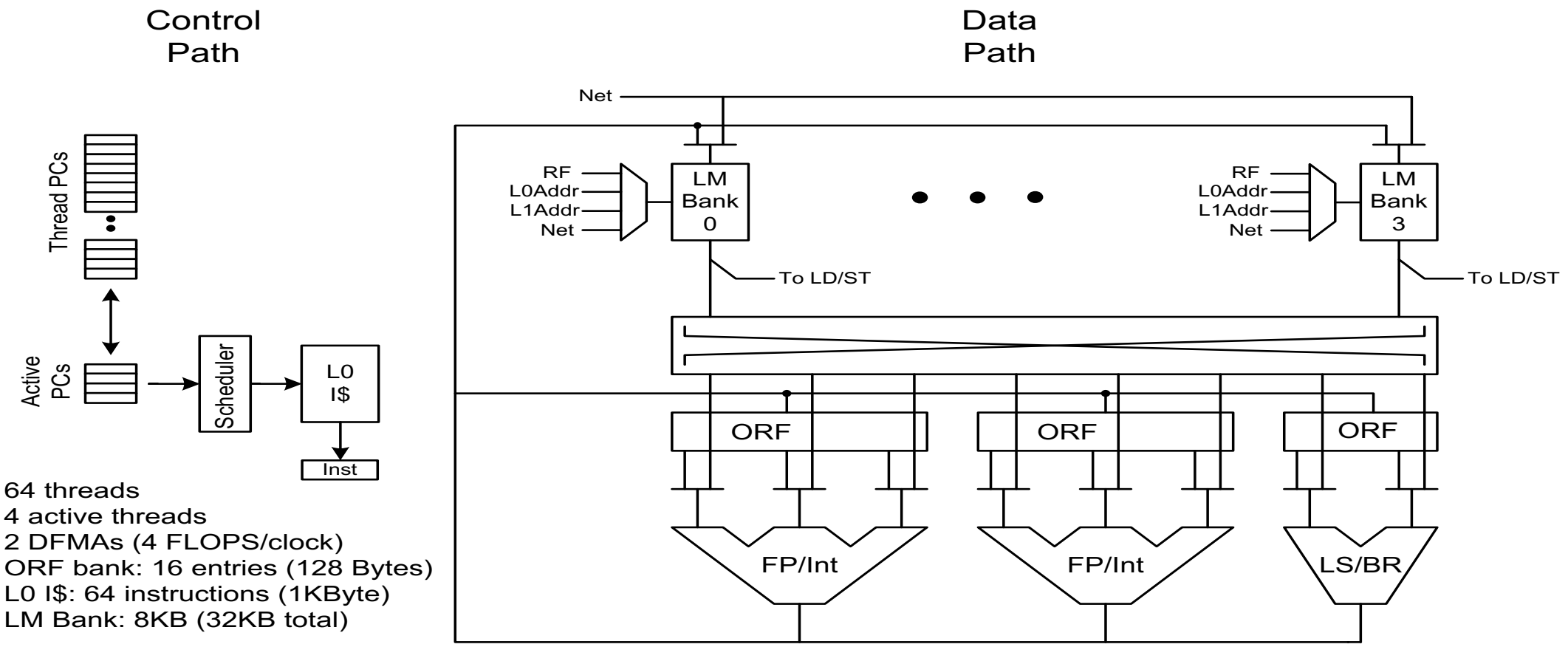
OOO Hi-perf



Natarajan [2003] (Alpha 21264)

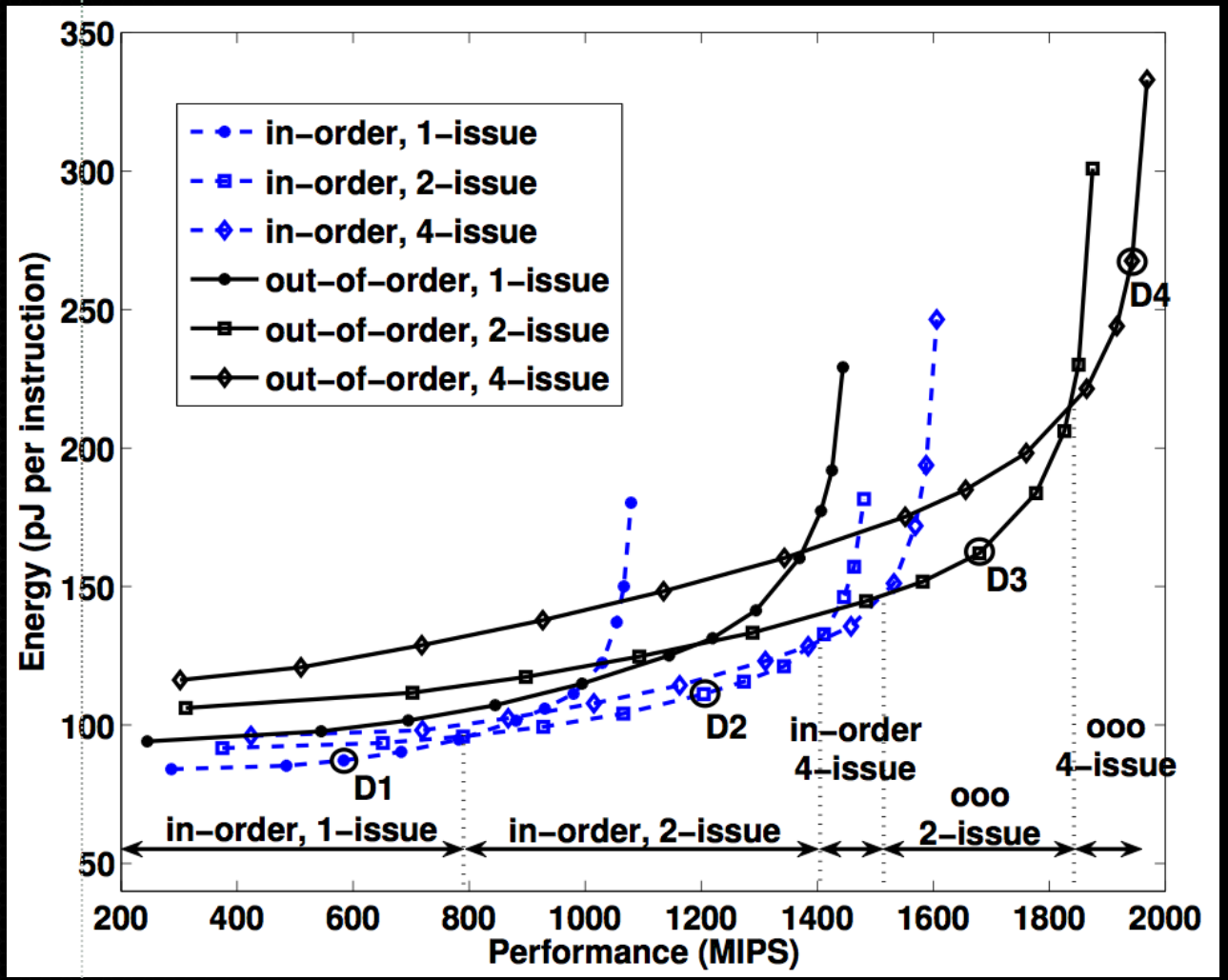



Pipeline Stages
IF
PC + I$ access begin
IF
I$ access complete
Instruction Issue (in-order)
Operand Read (out of order)
EX
Mem
WB
IF
ID
TLB (L1)
store
Prefetcher
TLB (L2)
MUX
L2 Cache
MUX
L1 Data Cache
ROB (Commit)
168 Entry
Load / Store Buffers
Load/Store
Reservation Station Buffer (3 Entry)
Branch/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

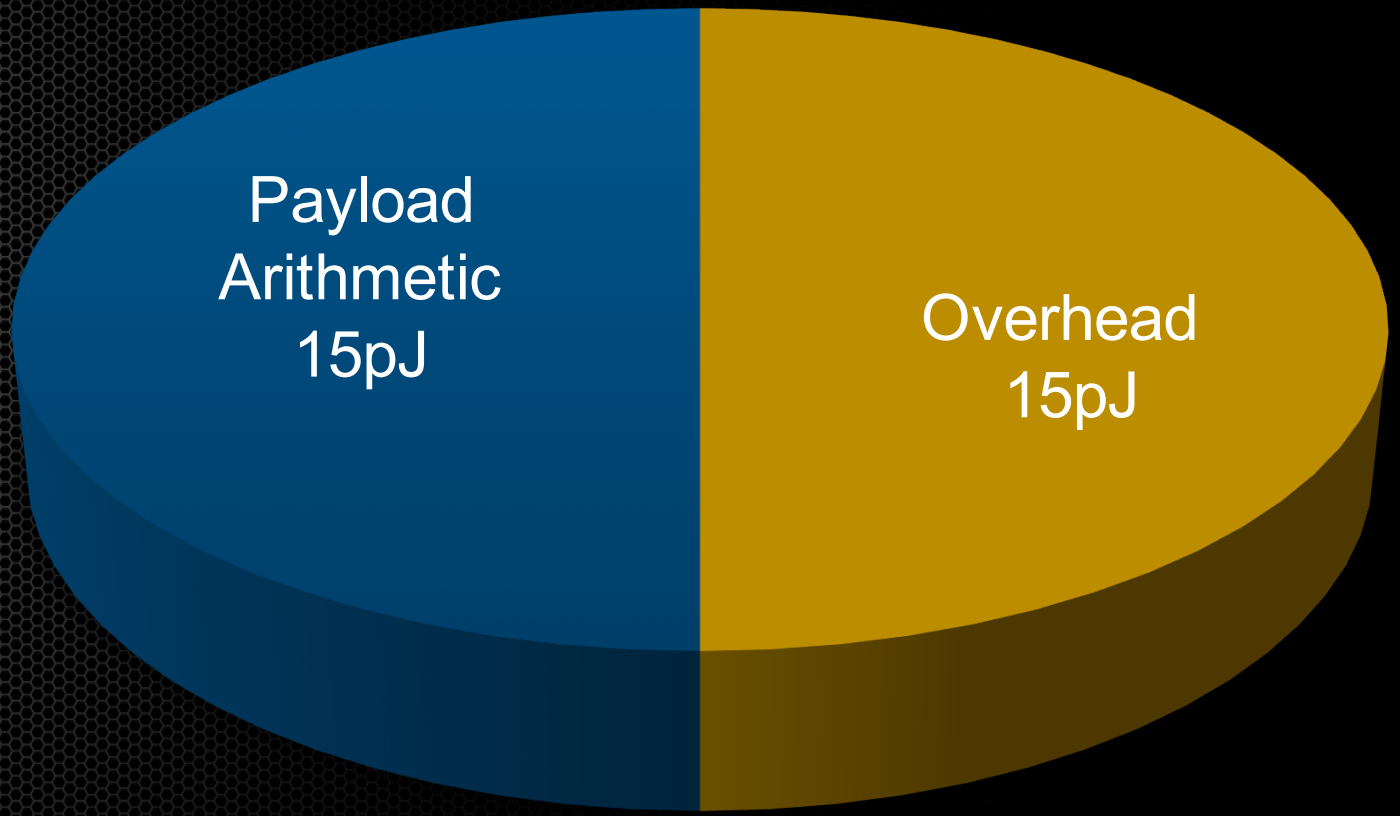


64 threads
4 active threads
2 DFMAs (4 FLOPS/clock)
ORF bank: 16 entries (128 Bytes)
L0 I$: 64 instructions (1KByte)
LM Bank: 8KB (32KB total)

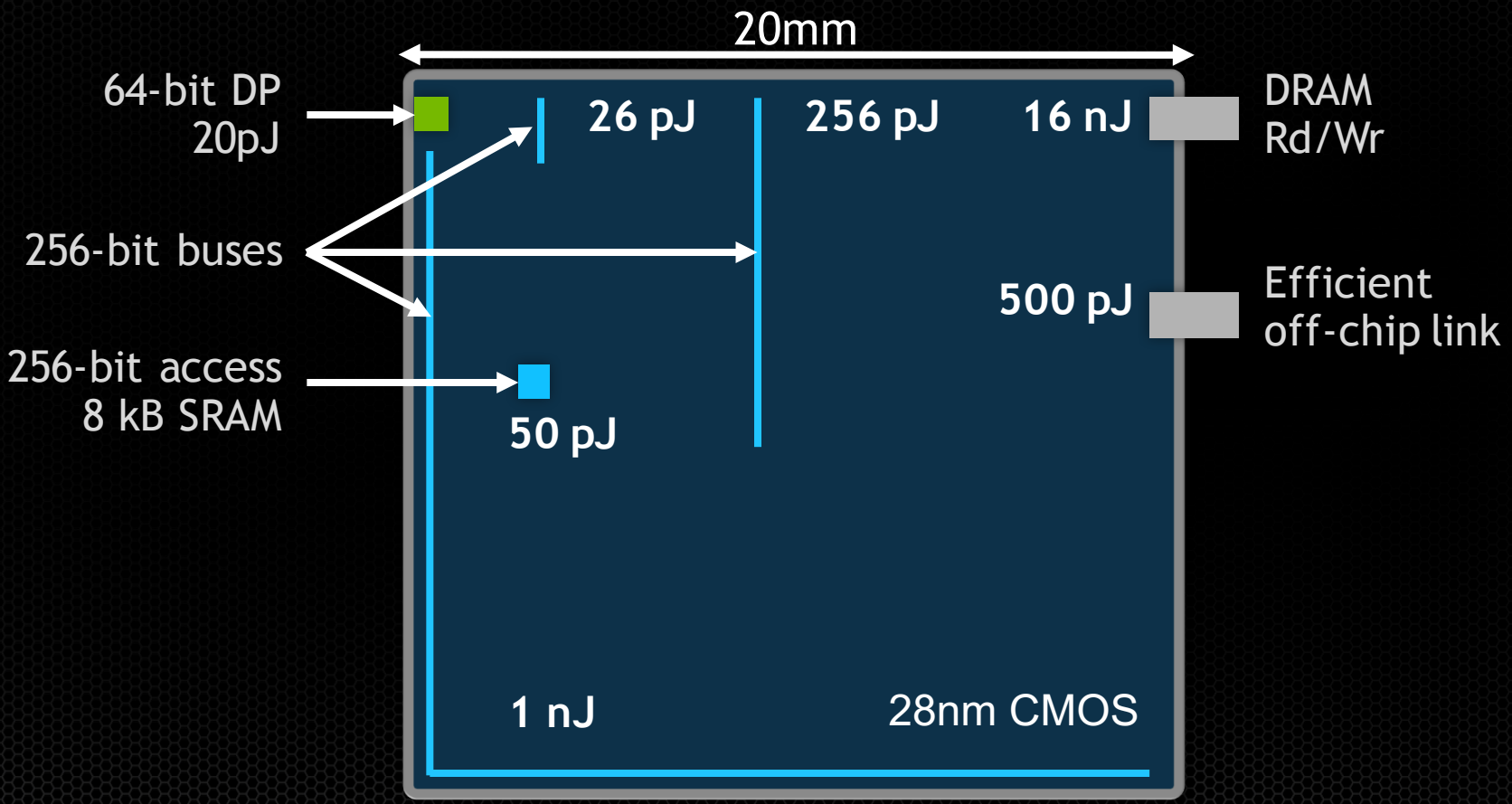
Simpler Cores = Energy Efficiency



Source: Azizi [PhD 2010]



Communication Dominates Arithmetic



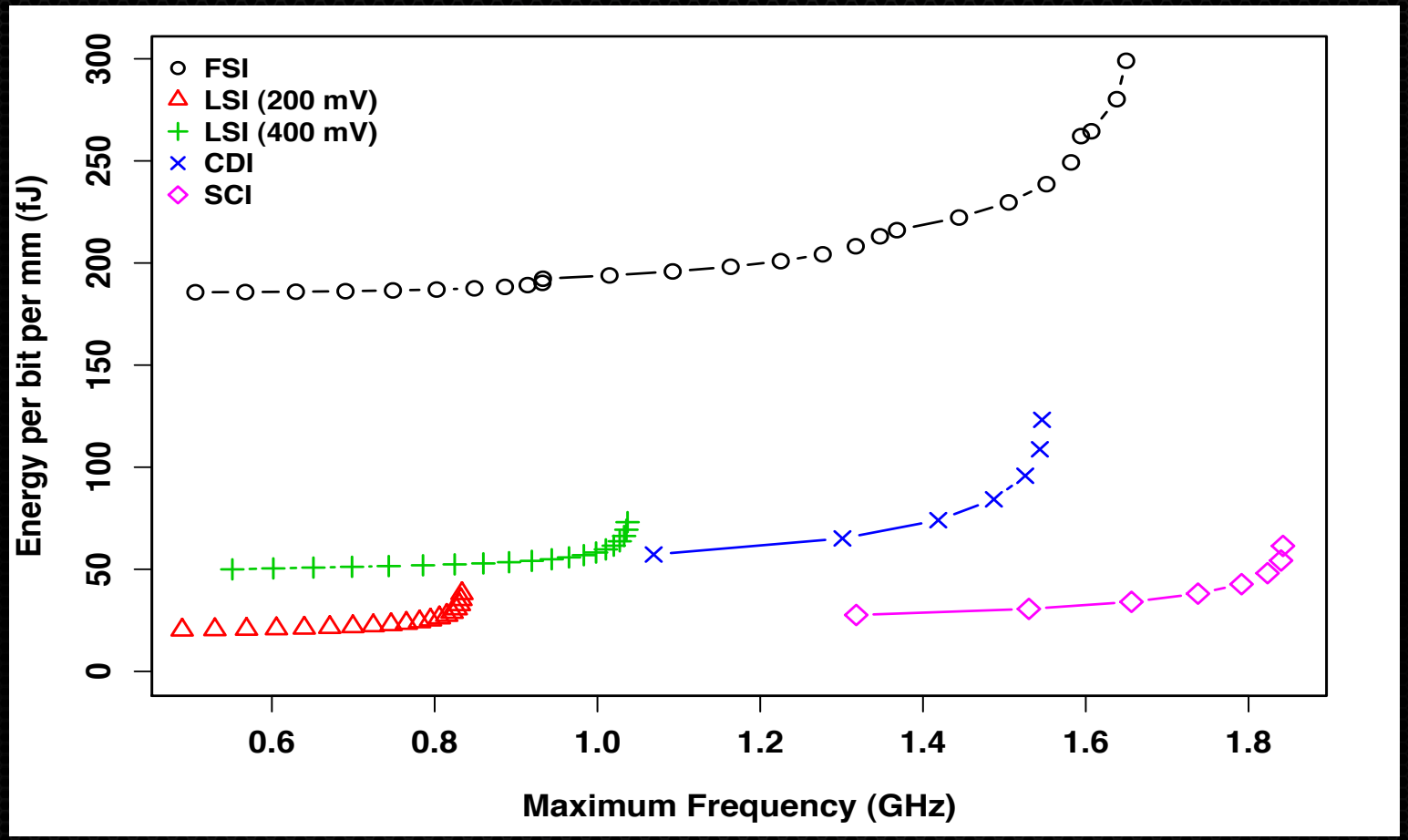
Energy Shopping List

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

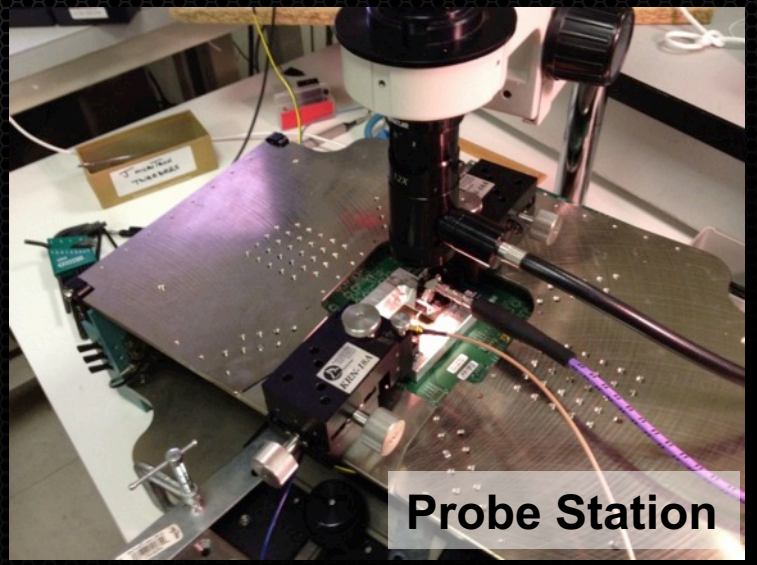
Memory Technology	45 nm	16nm
DRAM interface pin bandwidth	4 Gbps	50 Gbps
DRAM interface energy	20-30 pJ/bit	2 pJ/bit
DRAM access energy	8-15 pJ/bit	2.5 pJ/bit

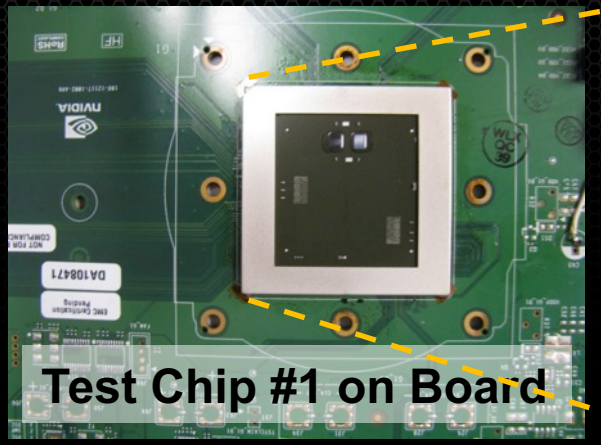
FP Op lower bound
=
4 pJ

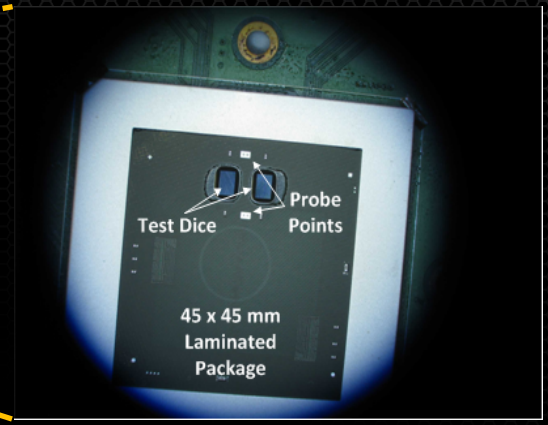
Keckler [Micro 2011], Vogelsang [Micro 2010]

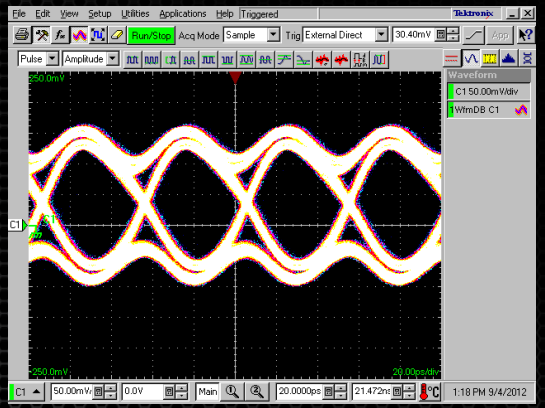

FSI
LSI (200 mV)
LSI (400 mV)
CDI
SCI
Energy per bit per mm (fJ)
0
50
100
150
200
250
300
0.6
0.8
1.0
1.2
1.4
1.6
1.8
Maximum Frequency (GHz)

GRS Test Chips

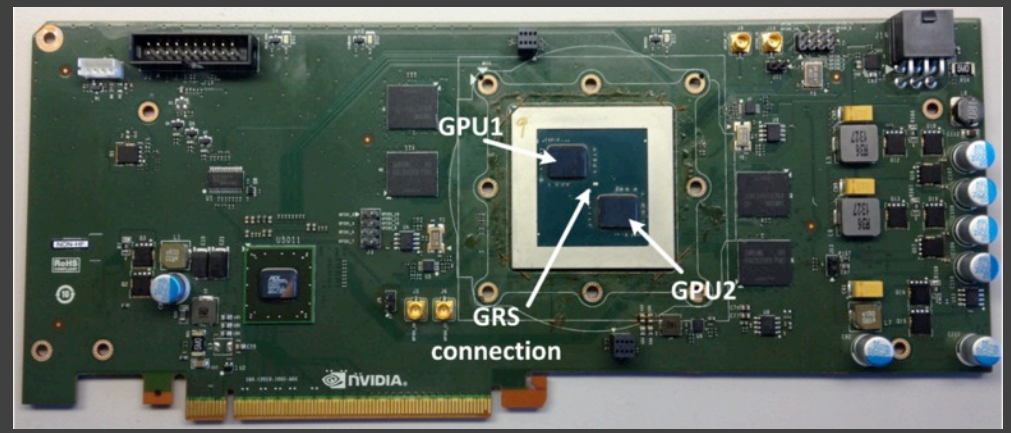








Eye Diagram from Probe

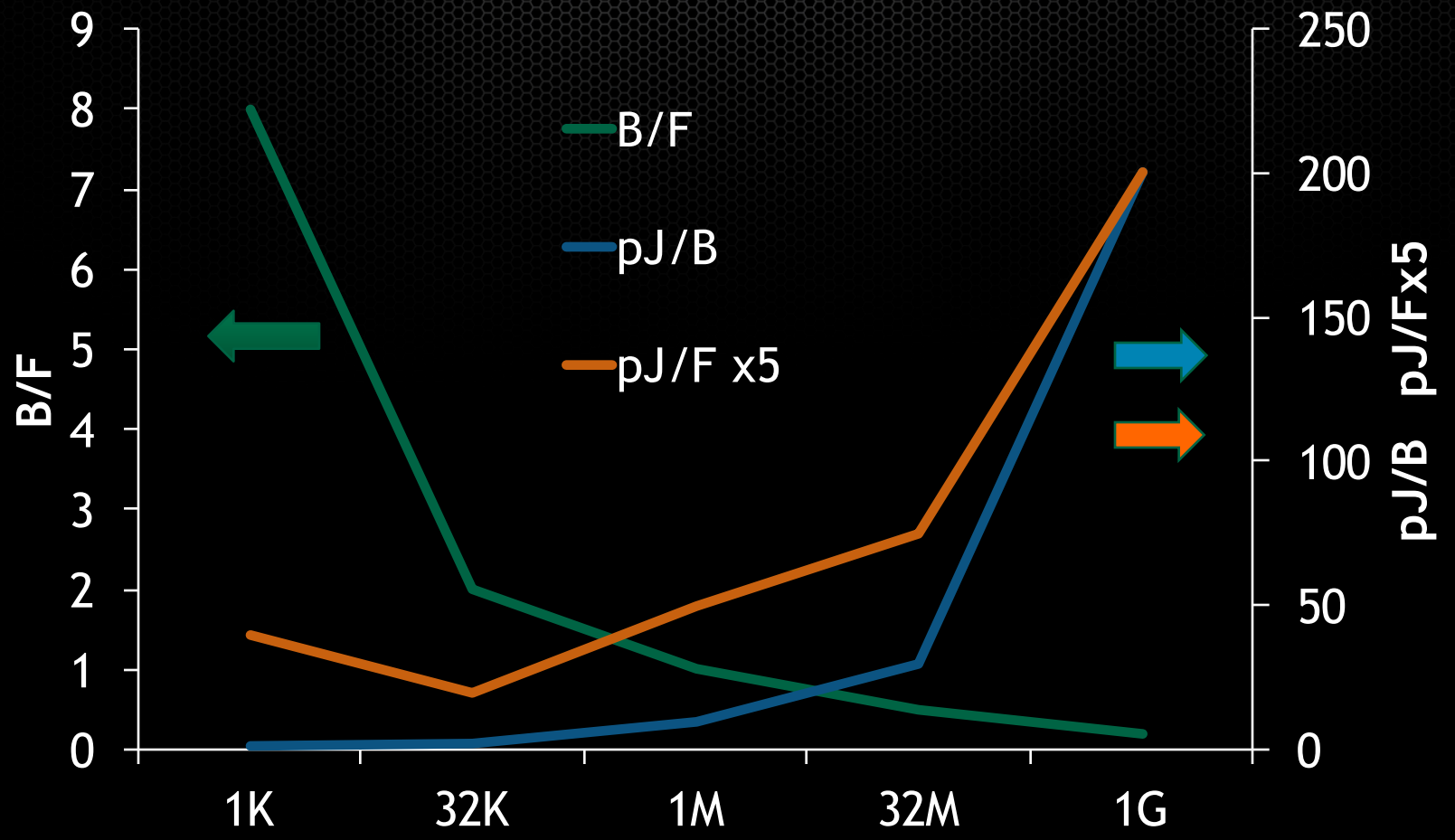


Test Chip #2 fabricated on production GPU

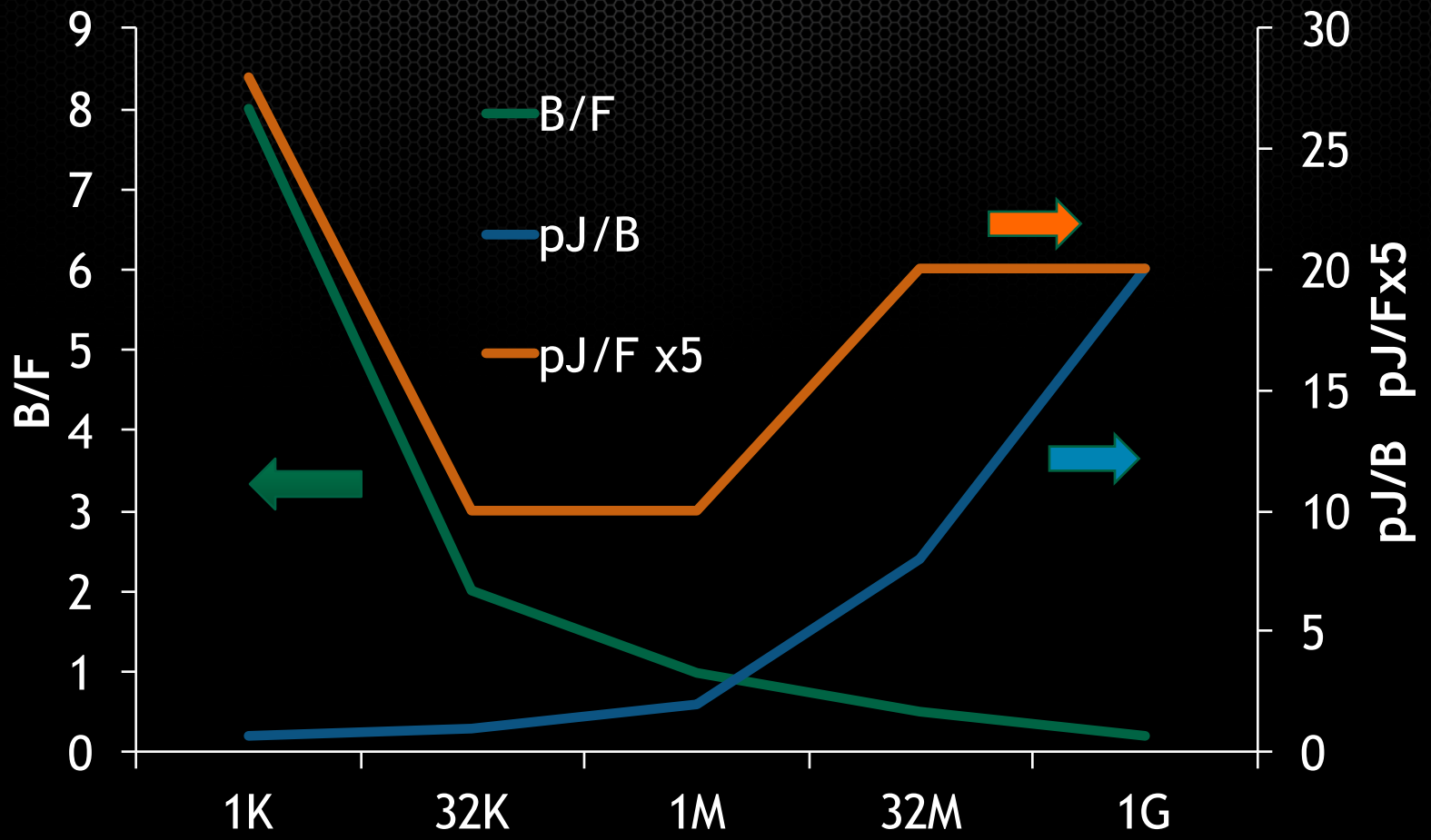
Poulton et al. ISSCC 2013, JSSCC Dec 2013

NVIDIA

The Locality Challenge
Data Movement Energy 77pJ/F



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



NVIDIA

2 Trends

The End of Dennard Scaling

Pervasive Parallelism

Processors aren’t getting faster just wider

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)

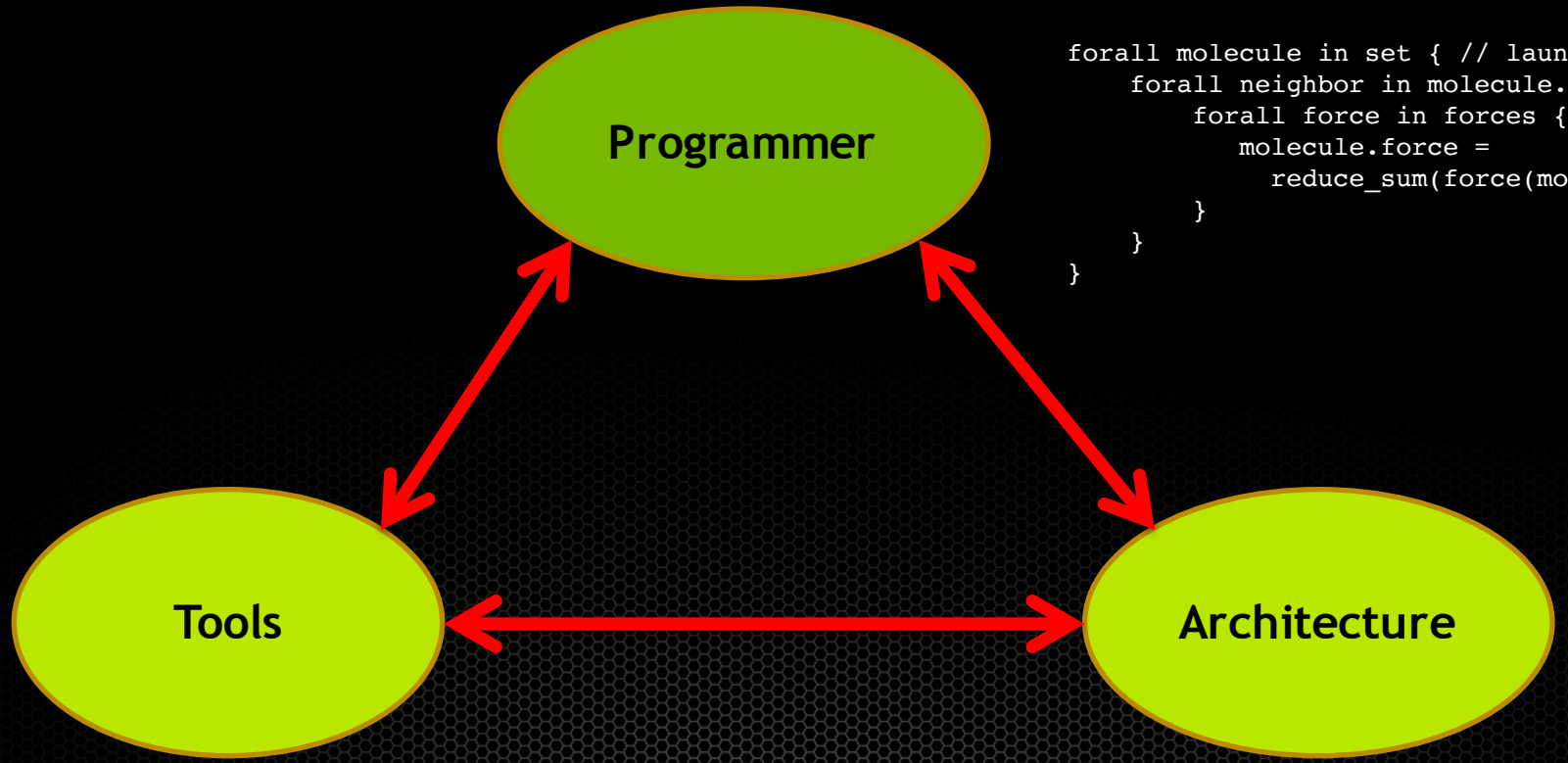
```
pid = fork() ; // explicitly managing threads

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

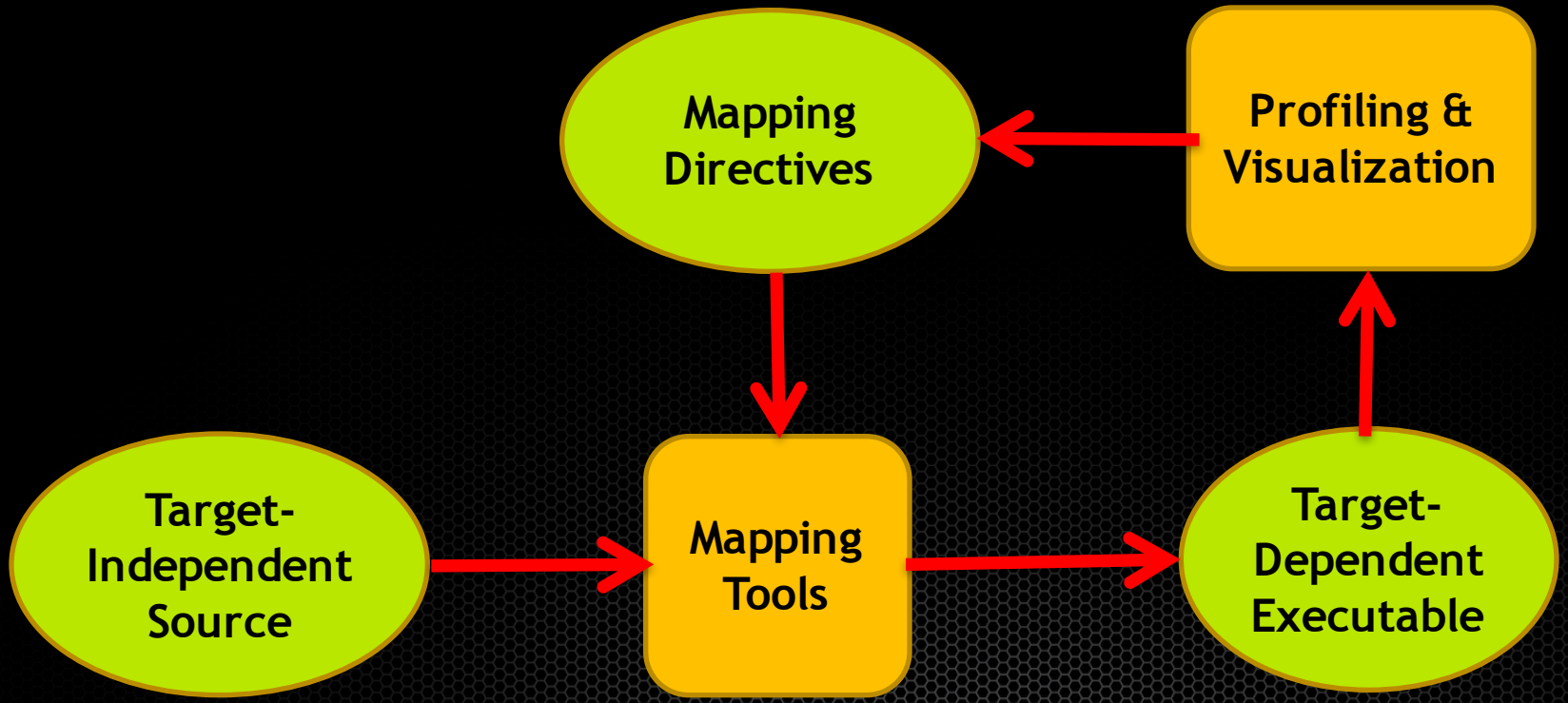
Programmers, tools, and architecture Need to play their positions

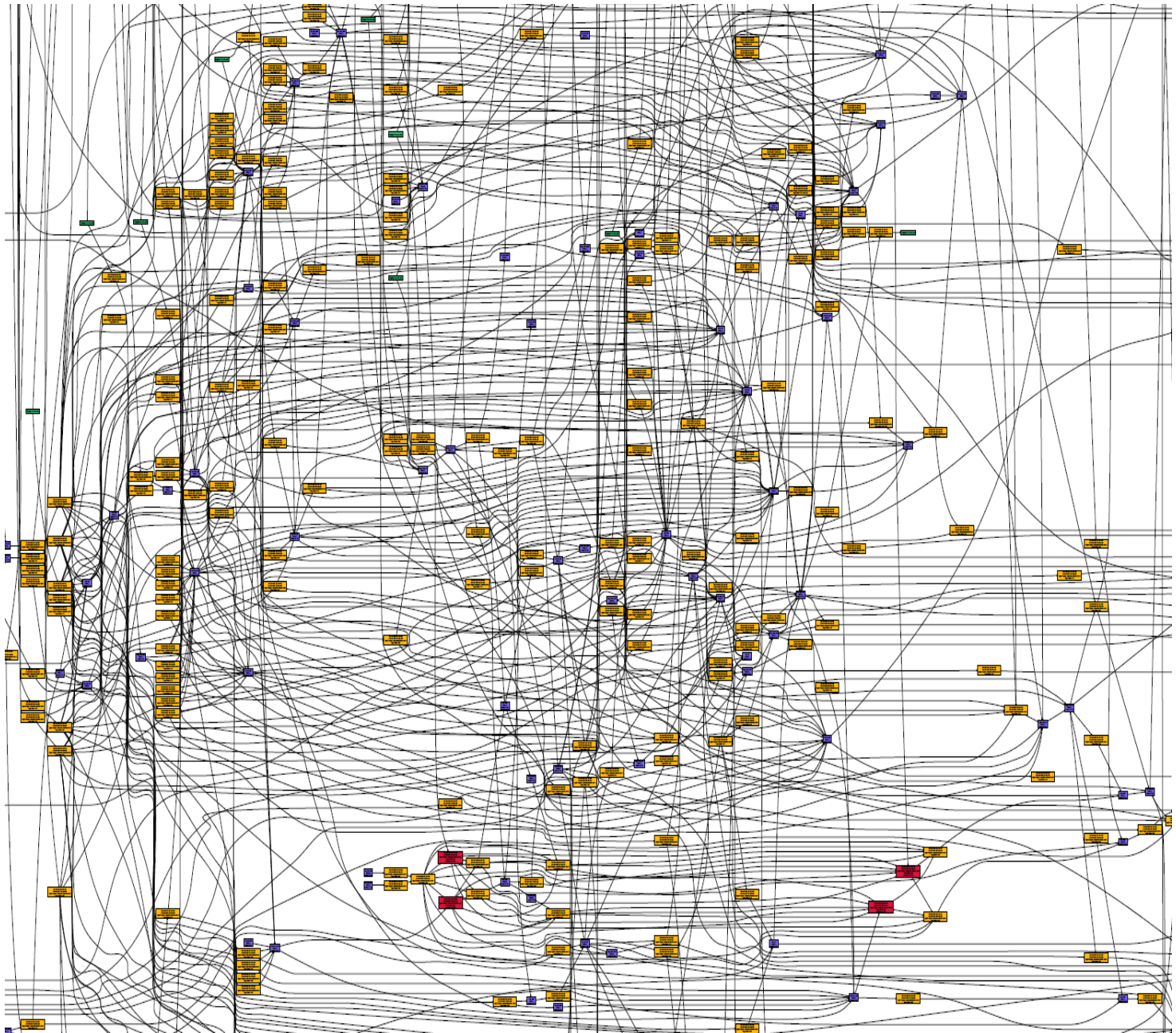



```
forall molecule in set { // launch a thread array
    forall neighbor in molecule.neighbors { //
        forall force in forces { // doubly nested
            molecule.force =
                reduce_sum(force(molecule, neighbor))
        }
    }
}
```

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

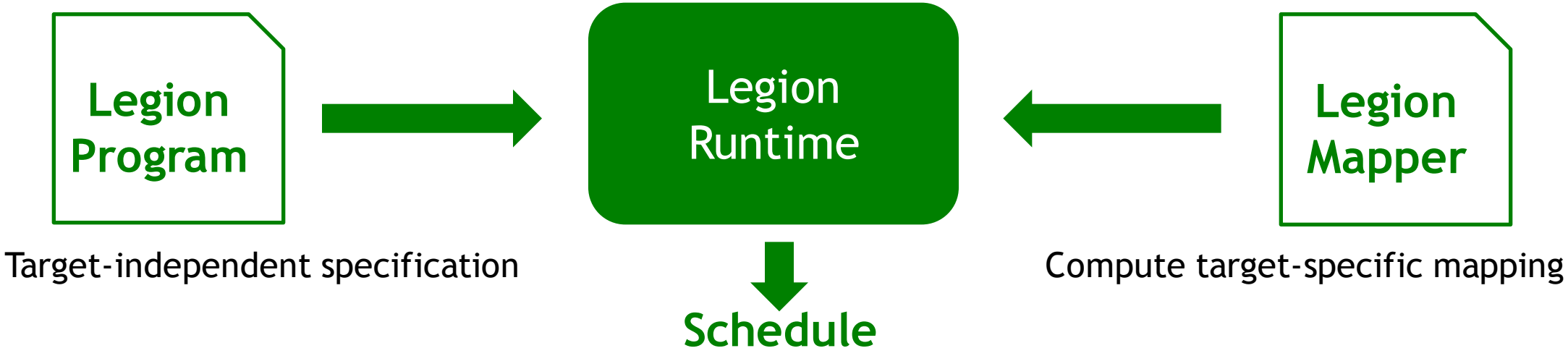
Exposed storage hierarchy
Fast comm/sync/thread mechanisms

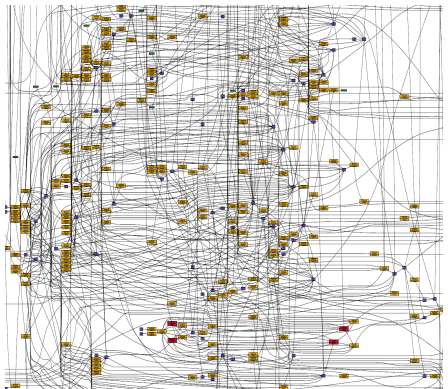

Mapping Directives
Profiling & Visualization
Target-Independent Source
Mapping Tools
Target-Dependent Executable



Legion Programming Model

- Separating program logic from machine mapping





The Legion Data Model: Logical Regions

Main idea: logical regions

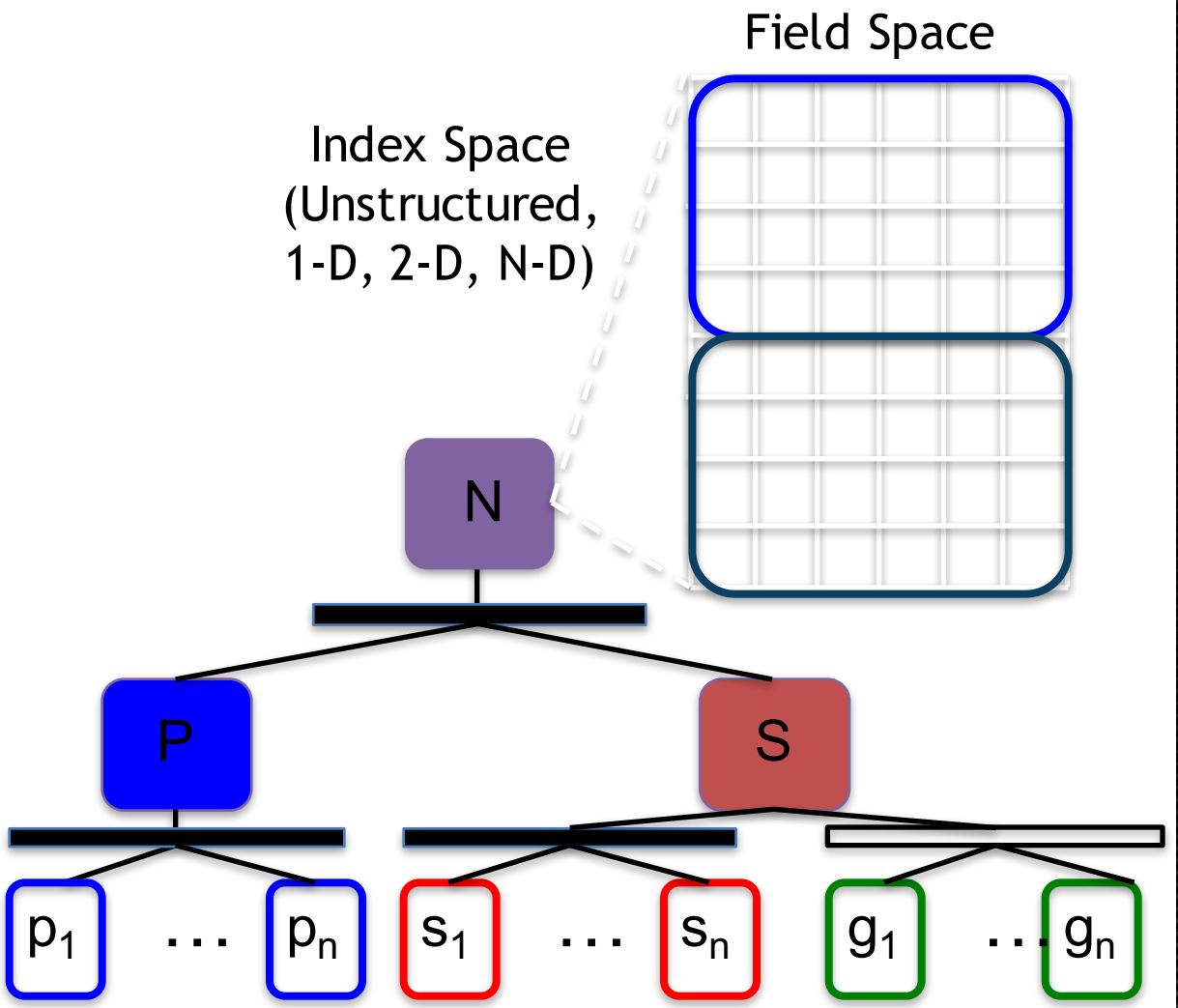
- Describe data abstractly
- Relational data model
- No implied layout
- No implied placement

Sophisticated partitioning mechanism

- Multiple views onto data

Capture important data properties

- Locality
- Independence/aliasing



The Legion Programming Model

Computations expressed as tasks

- Declare logical region usage
- Declare field usage
- Describe privileges:

read-only, read-write, reduce

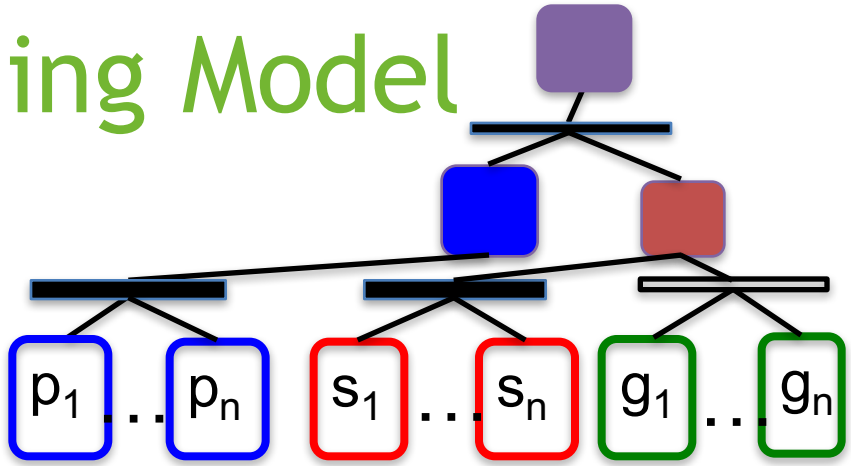
Tasks specified in sequential order

Legion infers implicit parallelism

Programs are machine-independent

- Tasks decouple computation
- Logical regions decouple

data

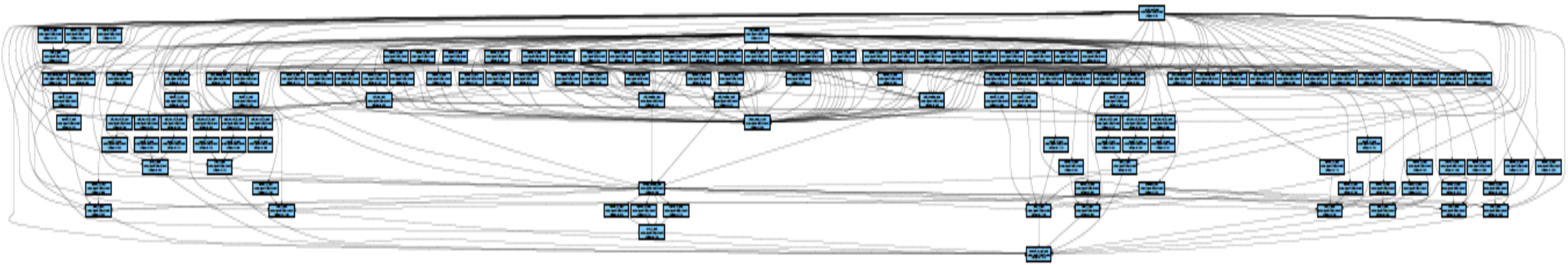


calc_currents(piece[0], p_0, s_0, g_0);

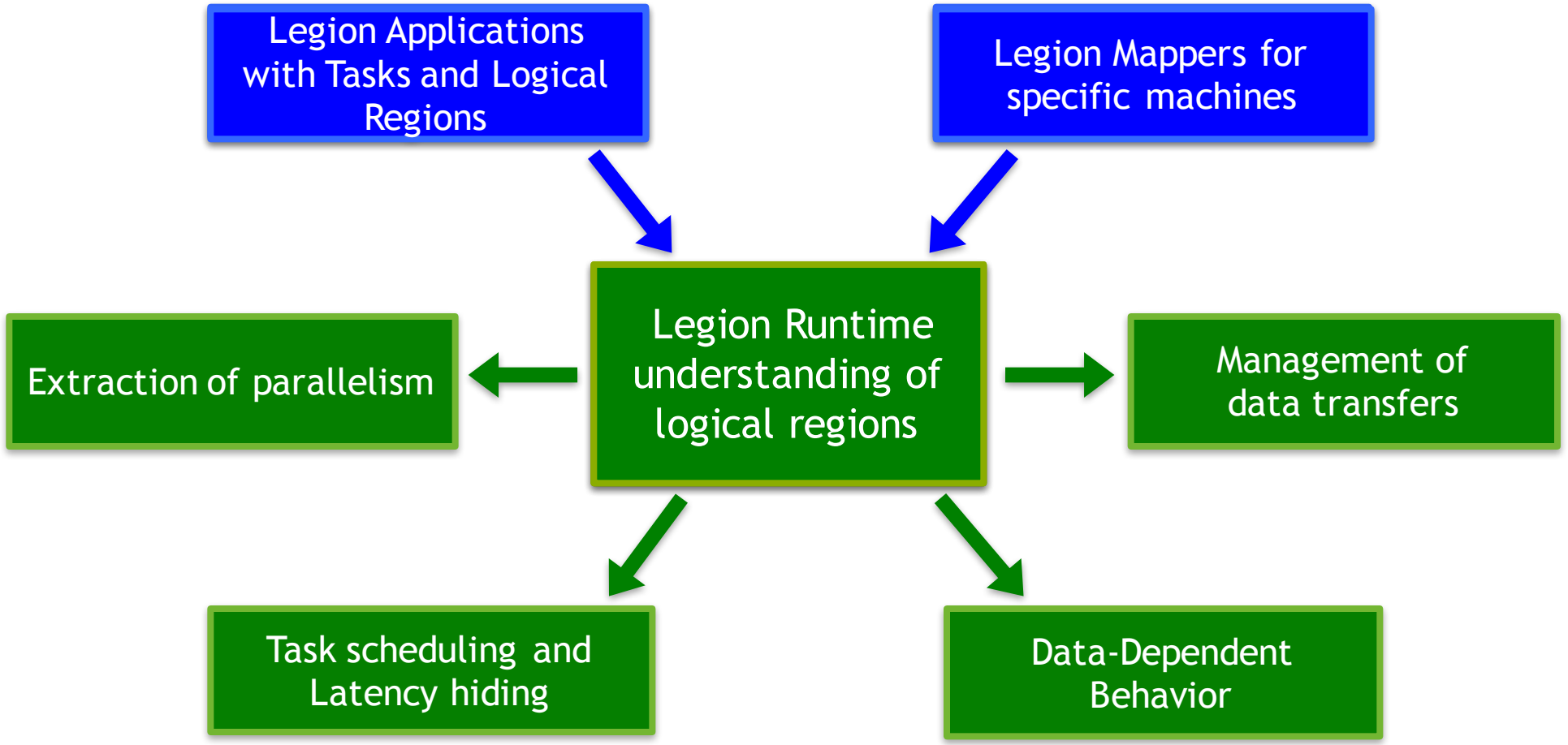
calc_currents(piece[1], p_1, s_1, g_1);

distribute_charge(piece[0], p_0, s_0, g_0);

distribute_charge(piece[1], p_1, s_1, g_1);



Legion Runtime System



Evaluation with a Real App: S3D

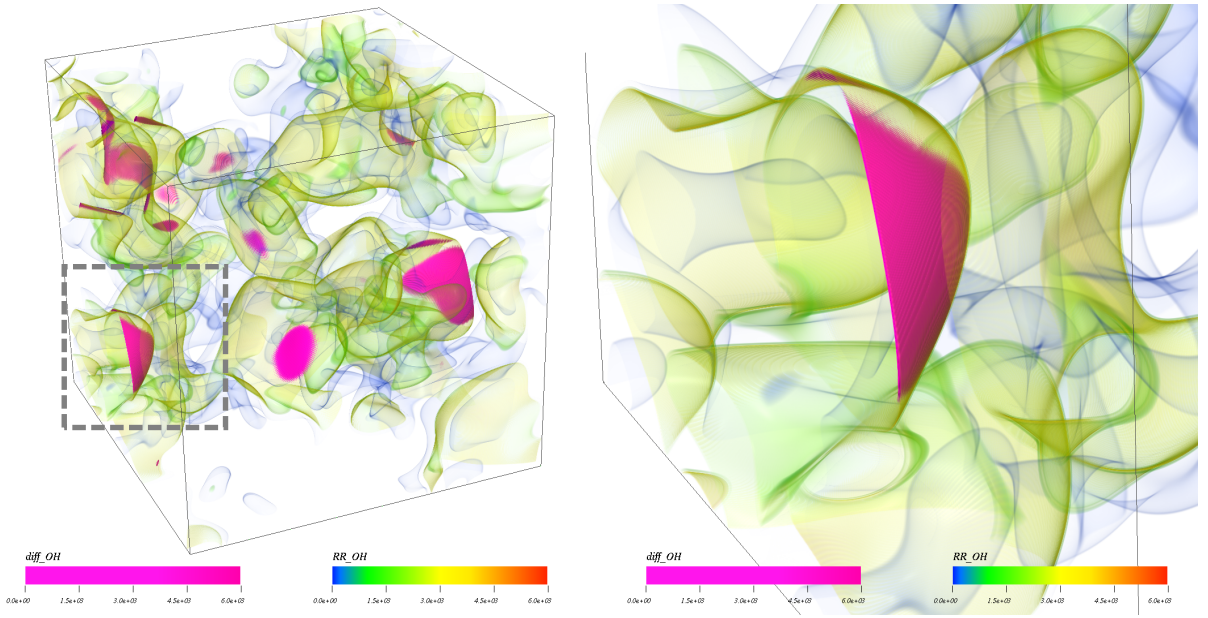
Evaluation with a production-grade combustion simulation

Ported more than 100K lines of MPI Fortran to Legion C++

Legion enabled new chemistry: Primary Reference Fuel (PRF) mechanism

Ran on two of the world's top 10 supercomputers for 1 month

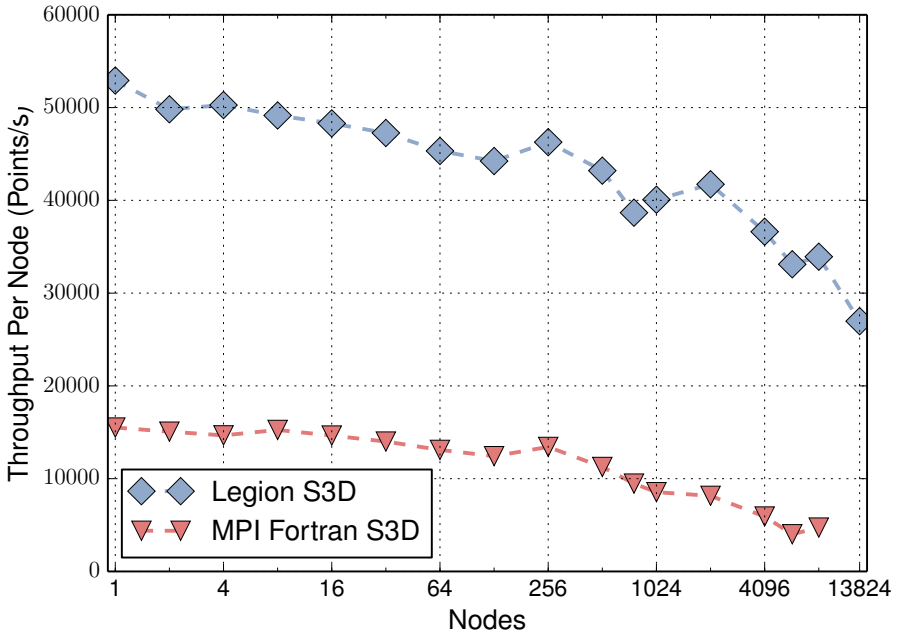
- Titan (#2) and Piz-Daint (#10)



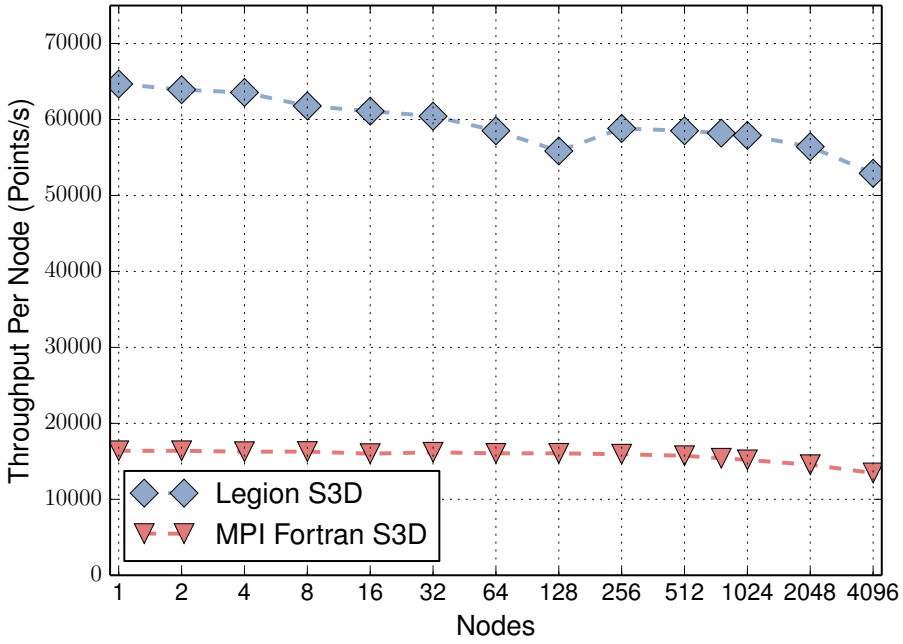
Performance Results: Original S3D

Weak scaling compared to vectorized MPI Fortran version of S3D

Achieved up to 6X speedup



Titan



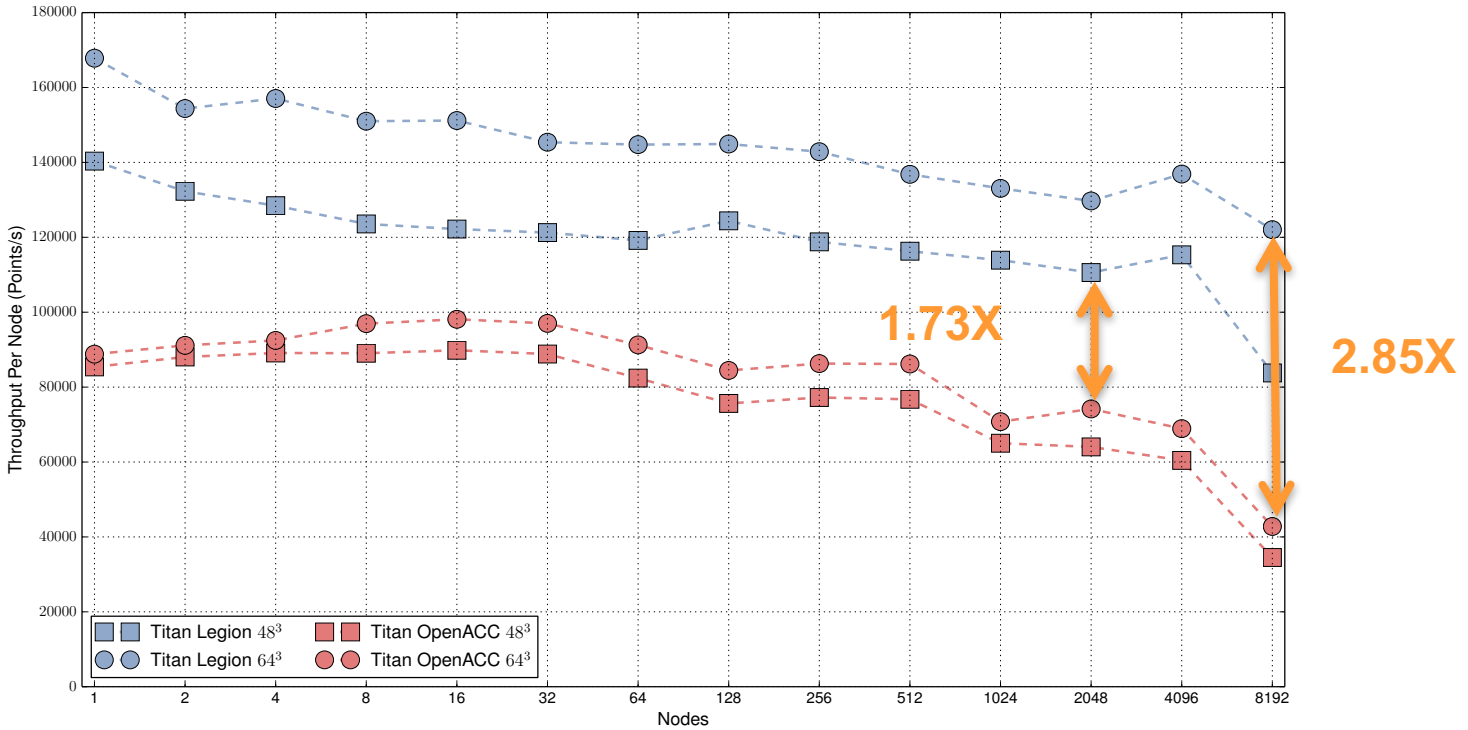
Piz-Daint

Performance Results: OpenACC S3D

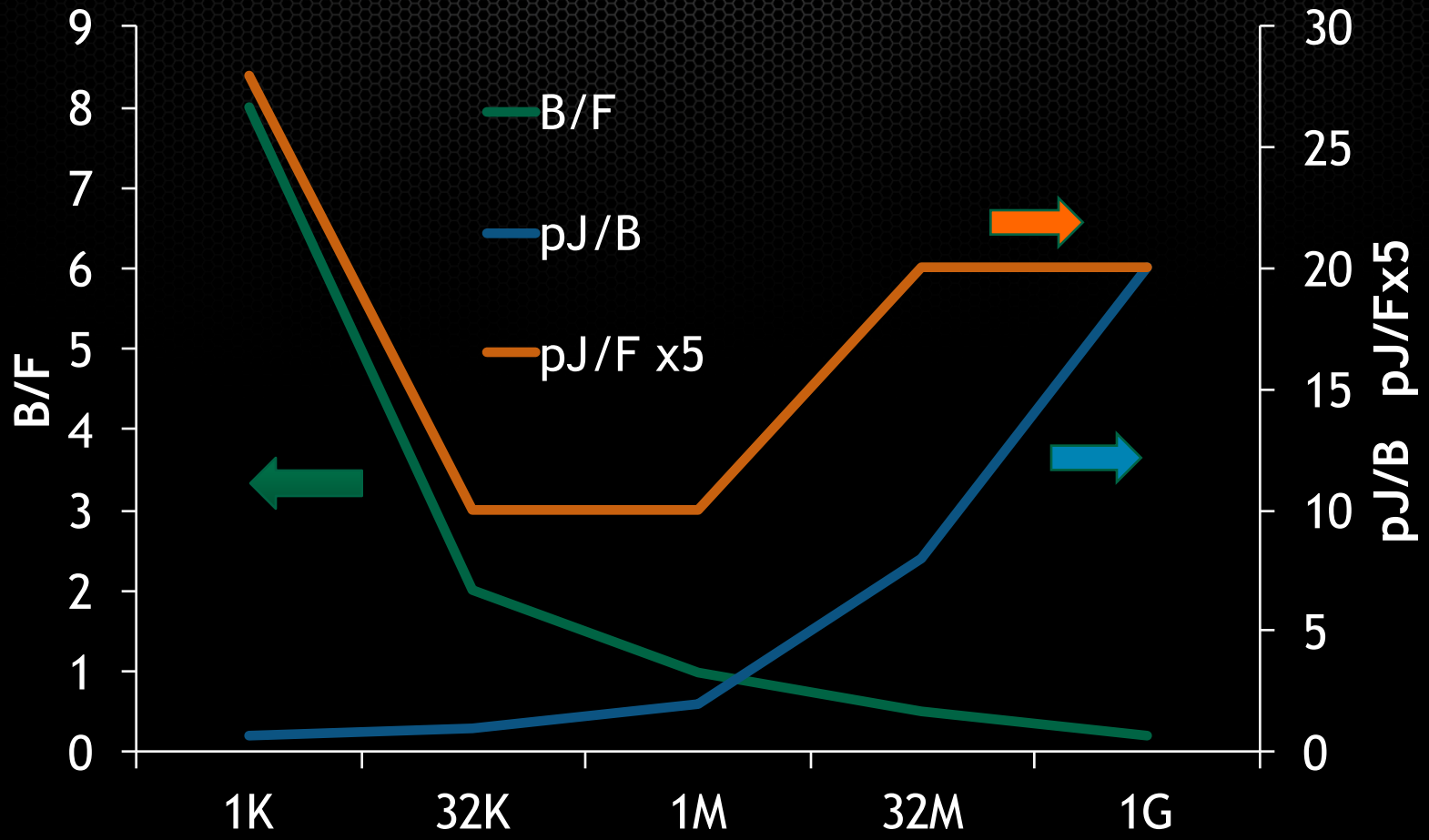
Also compared against experimental MPI+OpenACC version

Achieved 1.73 - 2.85X speedup on Titan

Why? Humans are really bad at scheduling complicated applications

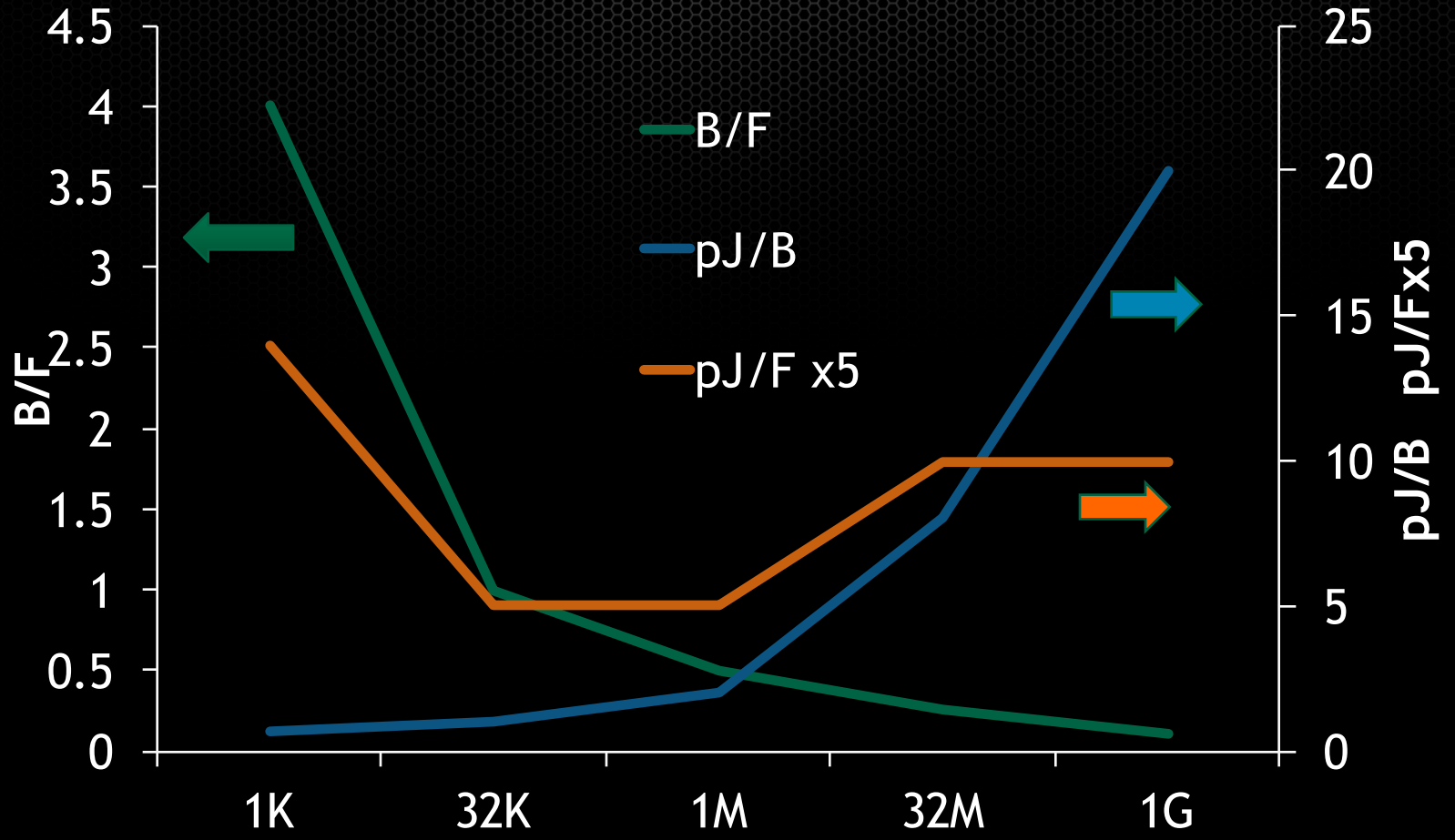


Optimized Circuits
77pJ/F -> 18pJ/F



NVIDIA

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
  - Legion demonstrates this on S3D
- Machine Learning is revolutionizing computing