The NVIDIA Kepler Accelerator and Long-term Futures

Steve Scott

Cray Technical Workshop on XK7 Programming

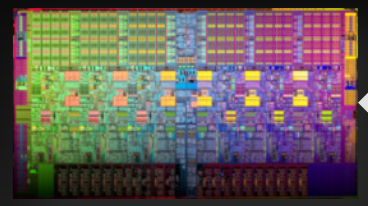
October 9, 2012

Power-Constrained Computing

Classic voltage scaling is over

$\Rightarrow$ chips have become power (not area) constrained
and it's getting worse with each generation

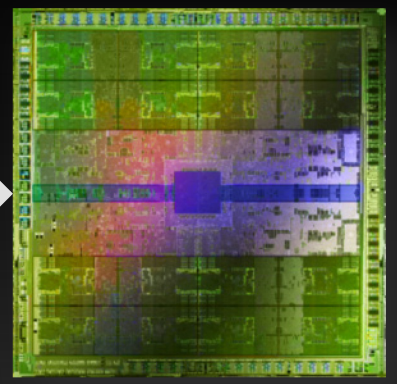
HPC Goes Hybrid



x86 CPU
Fast single threads
(serial work)

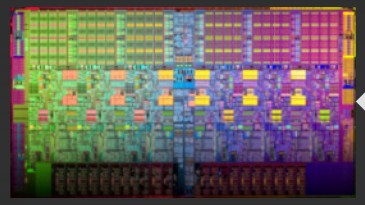
PCIe

Westmere
32nm
1.7 nJ/flop



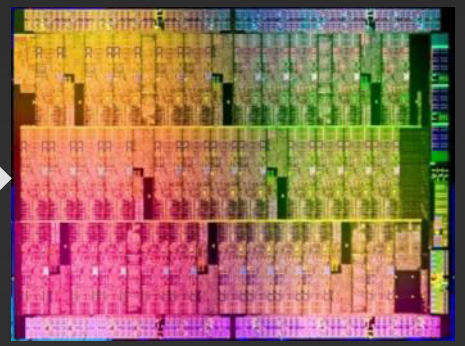
Fermi
40nm
225 pJ/flop

GPU
Extreme power-efficiency
(throughput work)



Xeon

PCIe

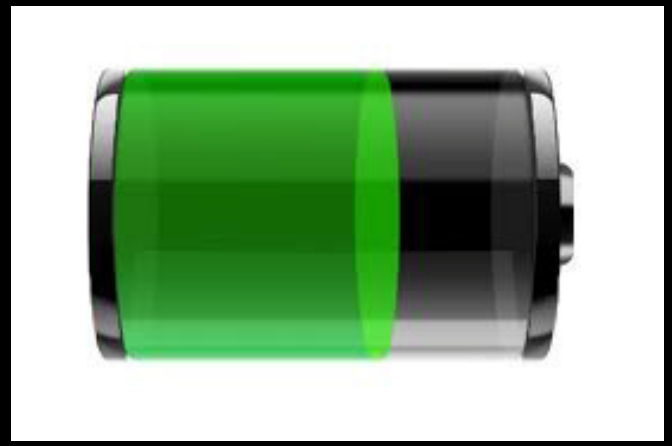


Intel MIC

(AMD Fusion too)

- Can't optimize for both single thread performance and power efficiency
- Must do most work with cores optimized for extreme energy efficiency
- Still need a few cores optimized for fast serial work

Overarching Goals for Tesla

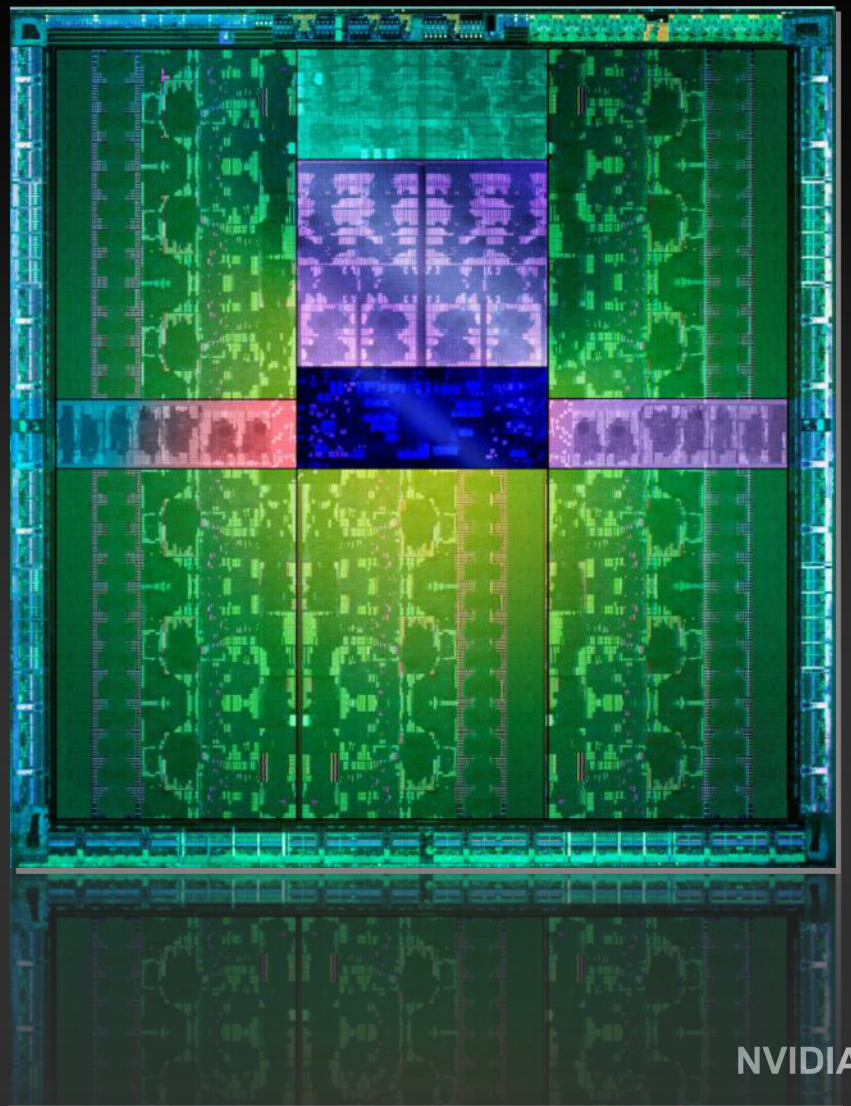


Power
Efficiency

Ease of
Programming
And Portability

Application
Space
Coverage

NVIDIA Confidential



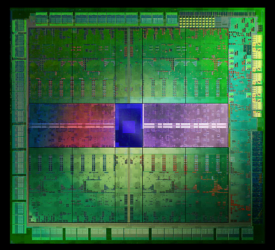
KEPLER

THE WORLD'S FASTEST, MOST EFFICIENT HPC ACCELERATOR

SMX	*(power efficiency)*
Hyper-Q	*(programmability and application coverage)*
Dynamic Parallelism	

NVIDIA Confidential

Tesla K10



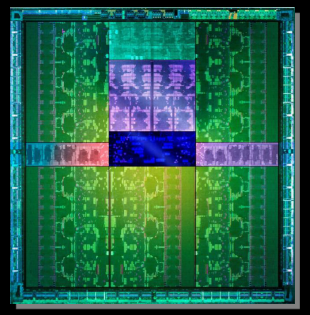
Dual GK104 GPUs

3x Single Precision

Video, Signal, Life Sciences, Seismic

Available Now

Tesla K20



GK110 GPU

3x Double Precision

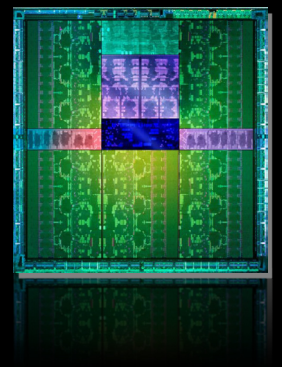
CFD, FEA, Finance, Physics, etc.

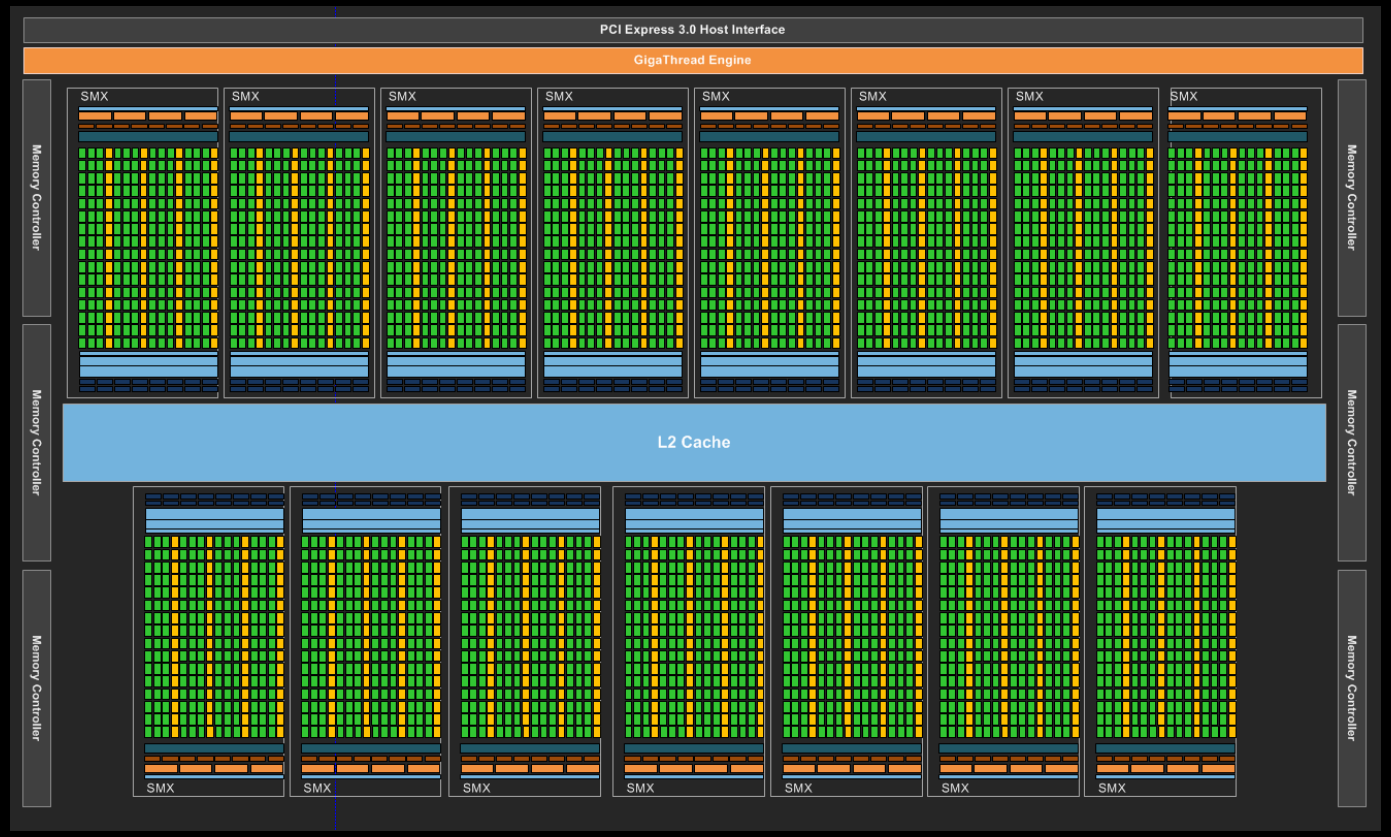
Available Q4 2012

NVIDIA Confidential

Kepler GK110 Block Diagram

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3





NVIDIA Confidential

Kepler GK110 SMX vs Fermi SM

3x sustained perf/W

Ground up redesign for perf/W

6x the SP FP units

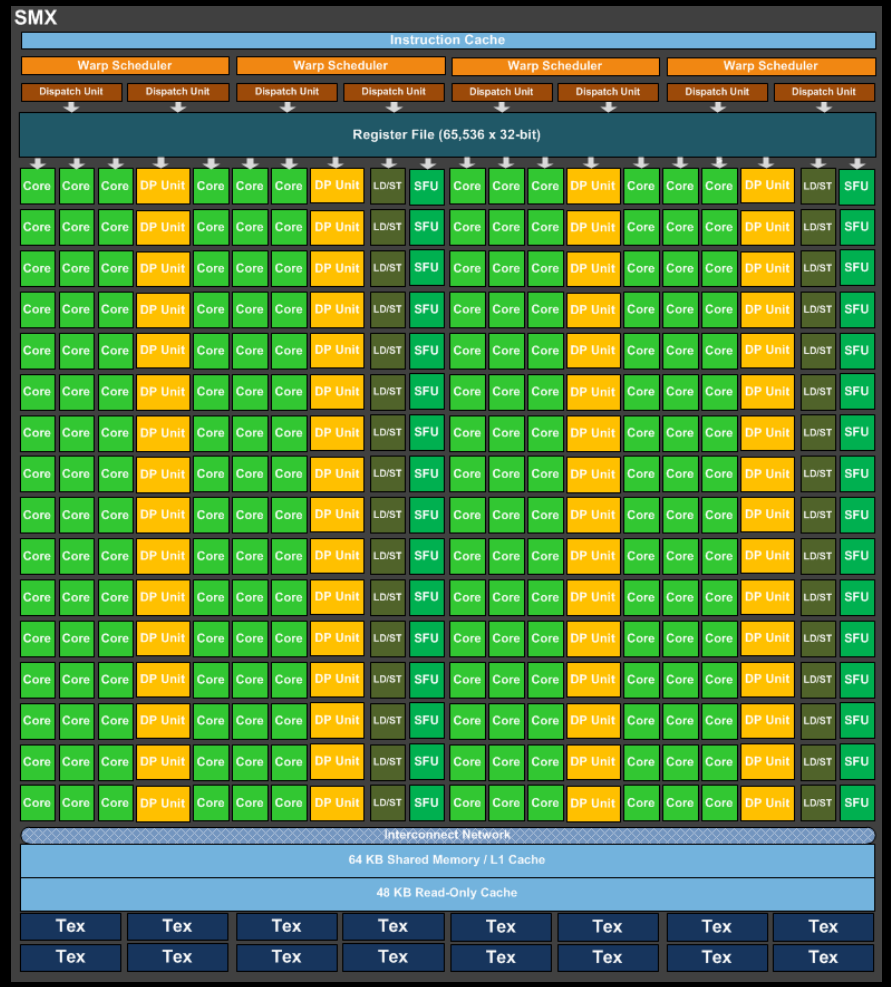
4x the DP FP units

Significantly slower FU clocks

~4x the overall instruction throughput

2x register file size

2x threadblocks & 1.33x threads



NVIDIA Confidential

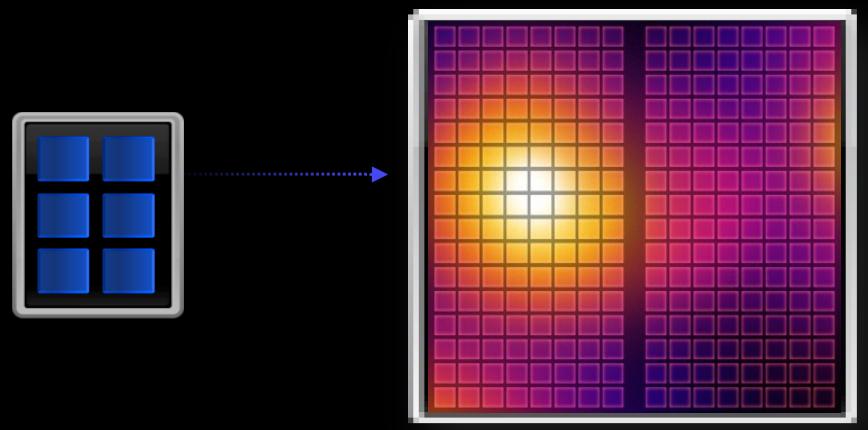
Selected Kepler ISA Enhancements

- Larger number of registers per thread
  - 63 in Fermi → 255 in Kepler
  - Common performance limited in Fermi due to register spilling
  - Significant performance improvement for some codes (e.g.: 5.3x on Quda QCD!)
- Atomic operations
  - Added int64 to match int32
  - Added functional units → 2-10x performance gains
- SHFL instruction for data exchange amongst threads of a warp
  - Broadcast, shifts, butterflies
  - Useful for sorts, reductions, etc.
- Loads through texture memory (48KB/SMX)
  - Extra bandwidth for read-only data (const__restrict)

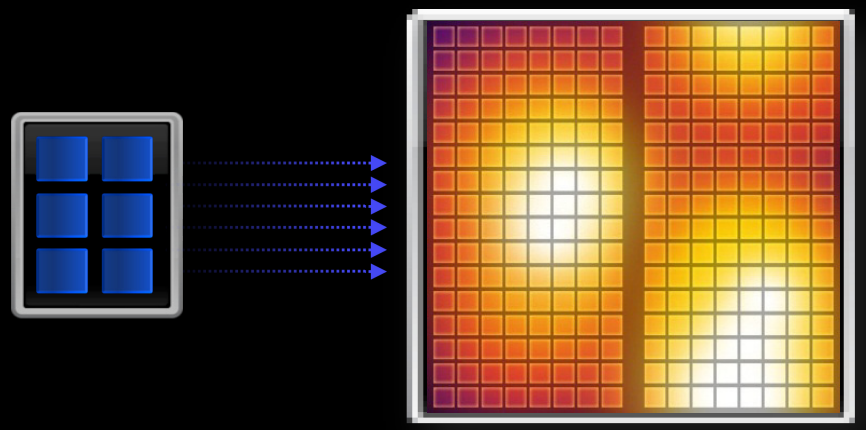
NVIDIA Confidential

Hyper-Q

FERMI
1 Work Queue

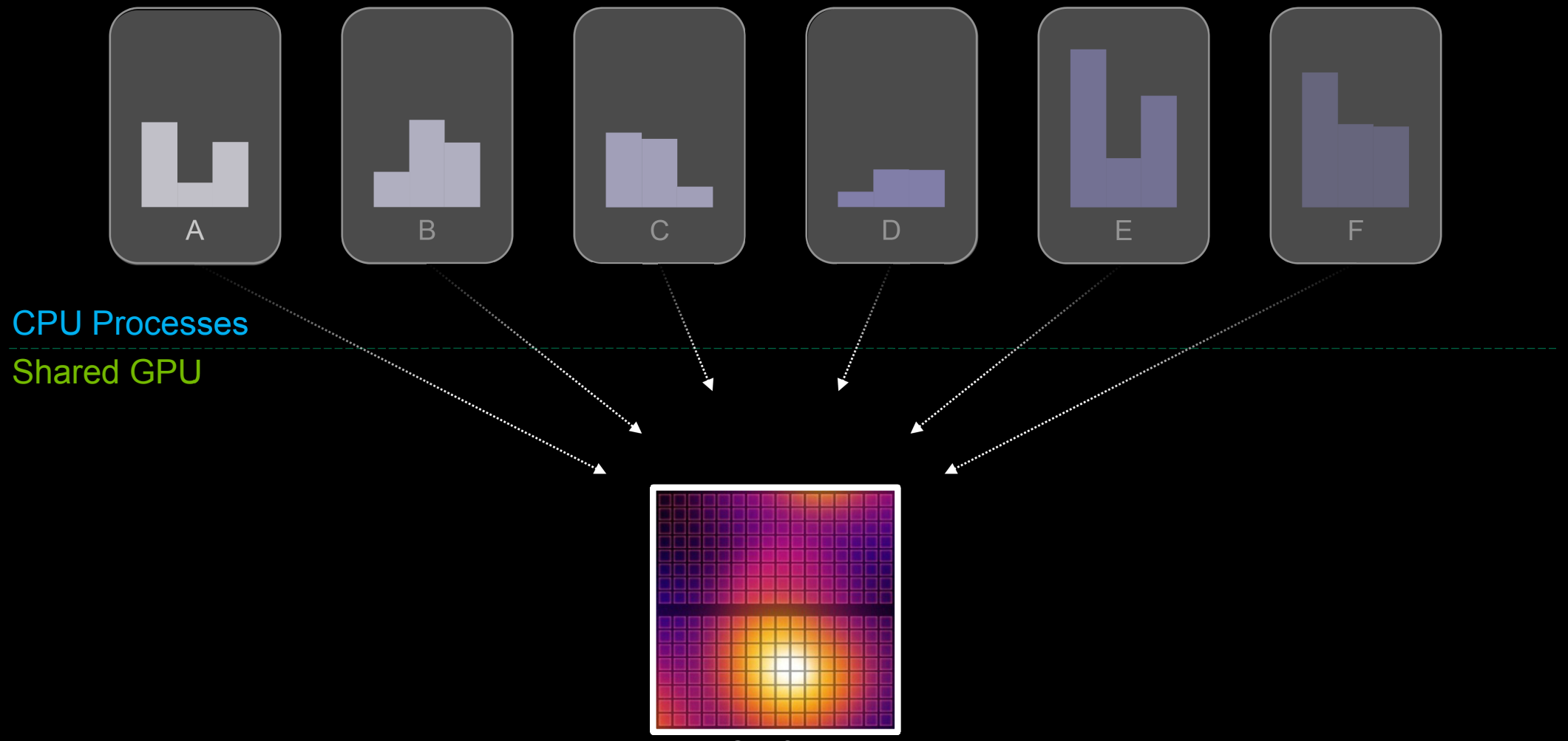


KEPLER
32 Concurrent Work Queues



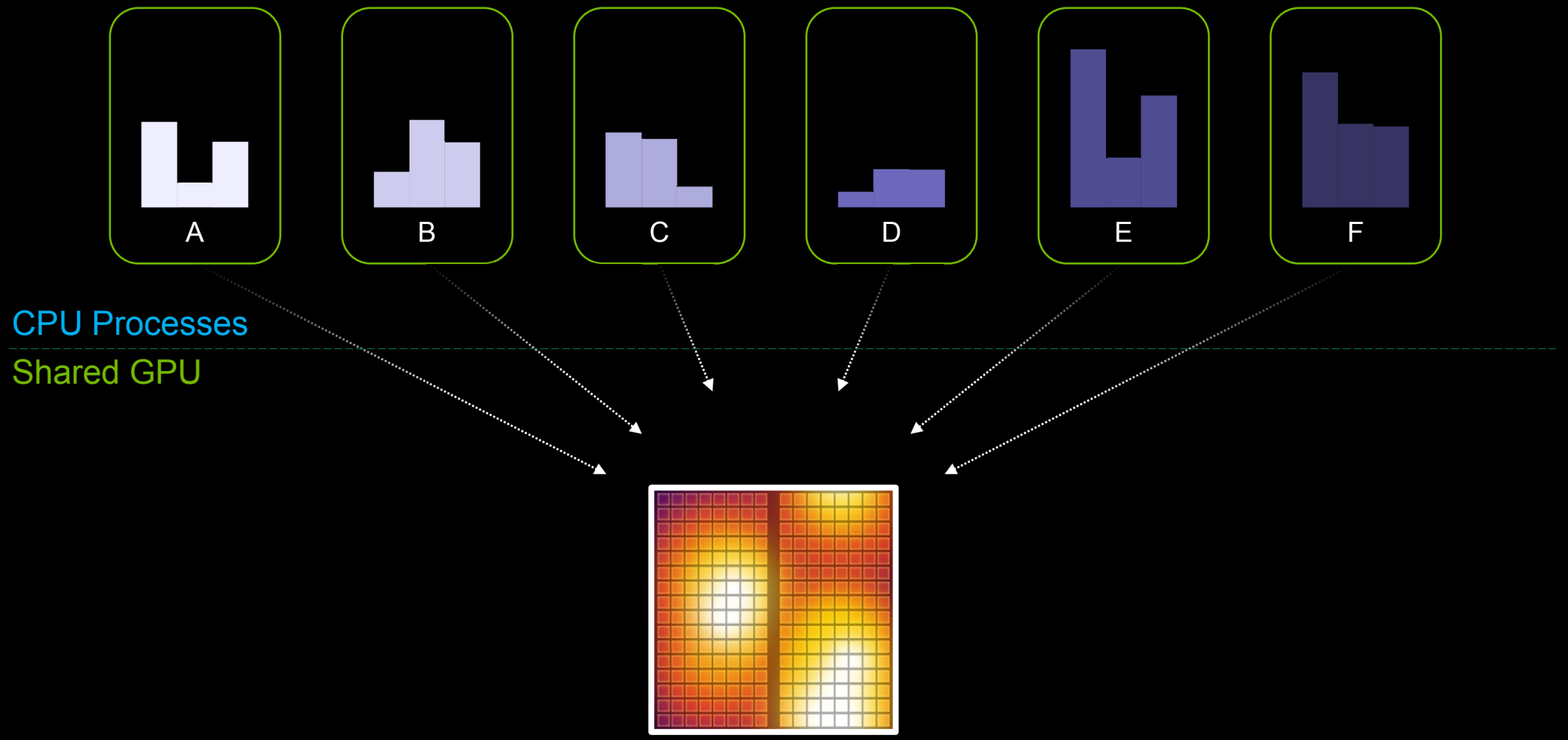
NVIDIA Confidential

Fermi: Time-Division Multiprocess



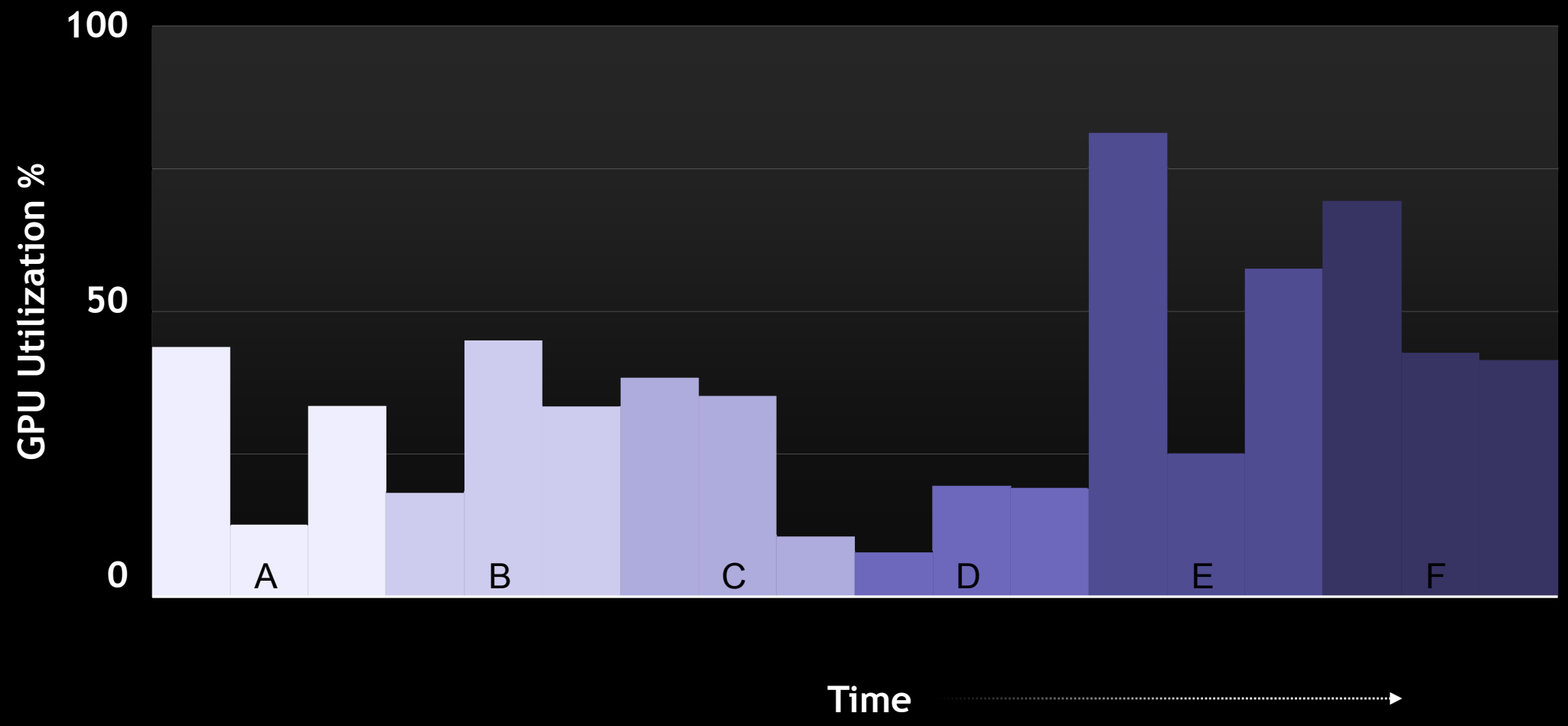
NVIDIA Confidential

Kepler Hyper-Q: Simultaneous Multiprocess



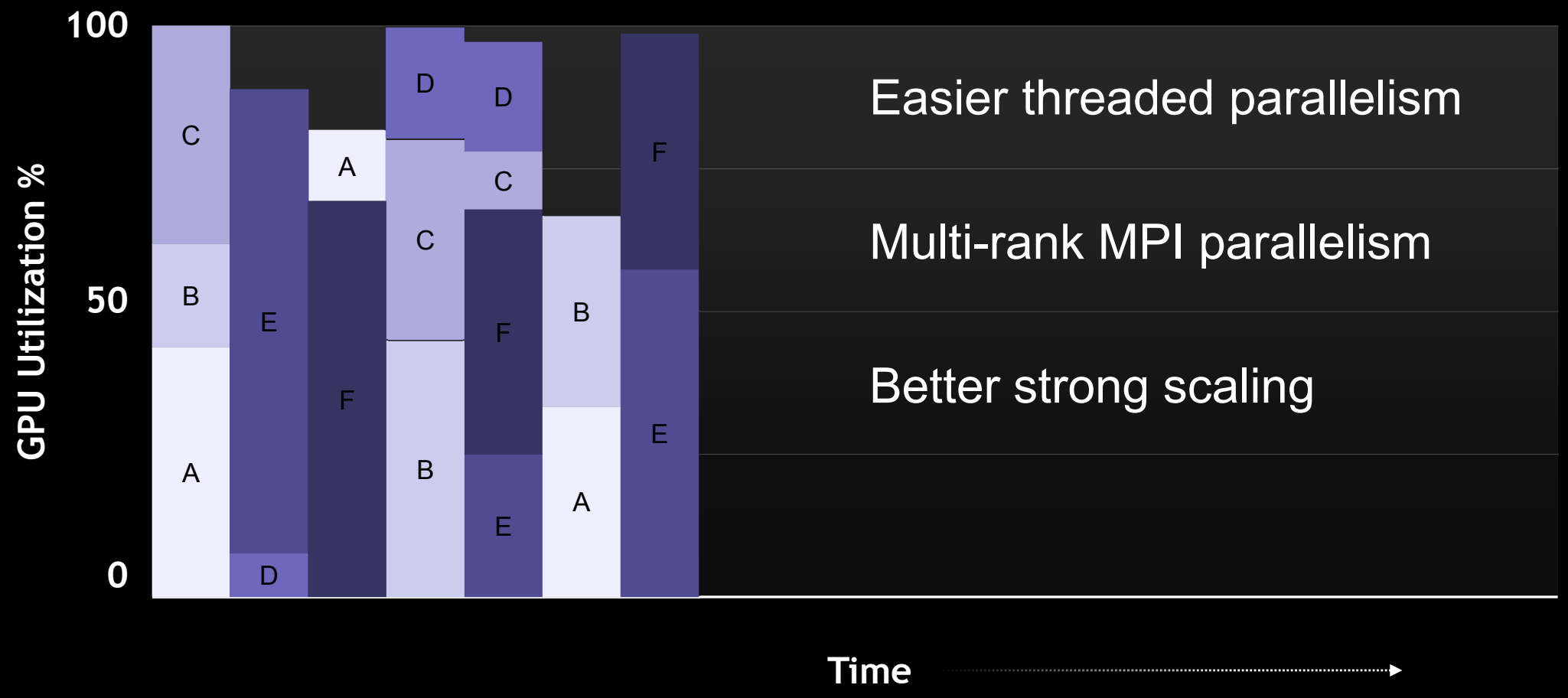
NVIDIA Confidential

Without Hyper-Q



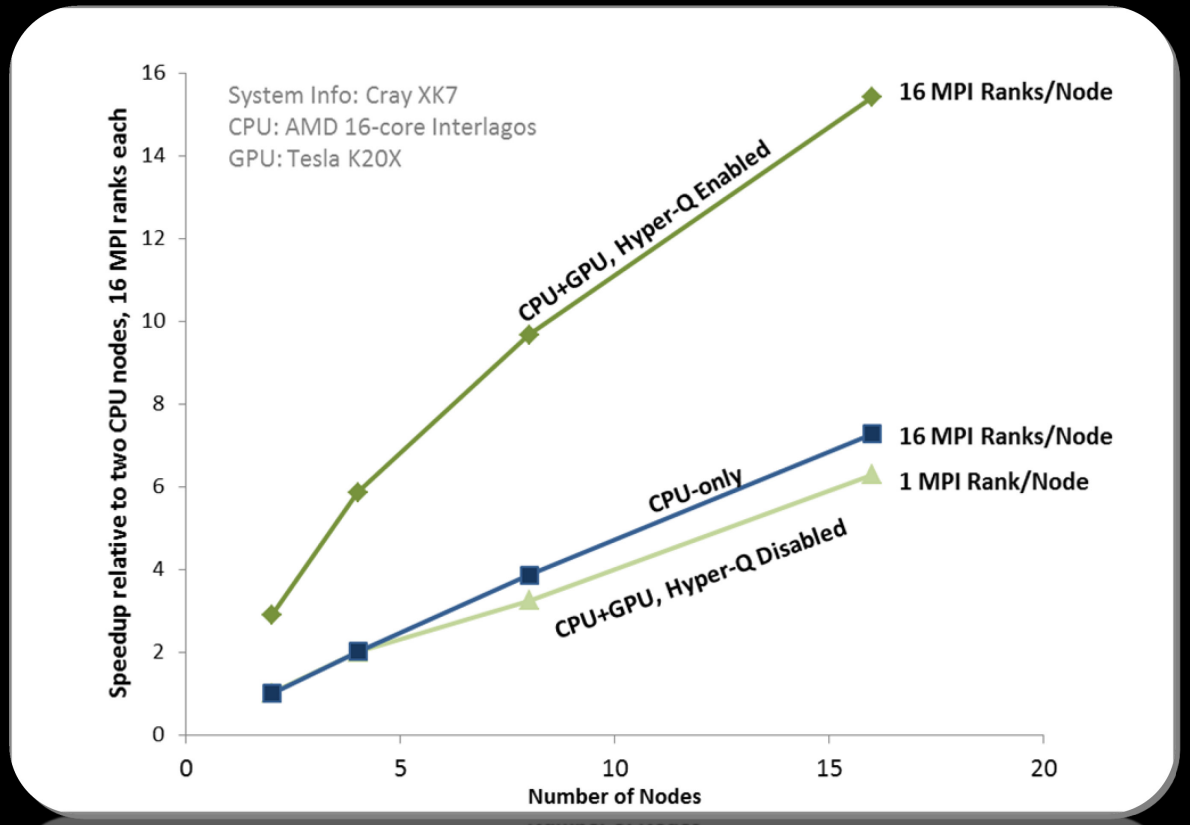
NVIDIA Confidential

With Hyper-Q



NVIDIA Confidential

Hyper-Q: Accelerating Legacy MPI Codes



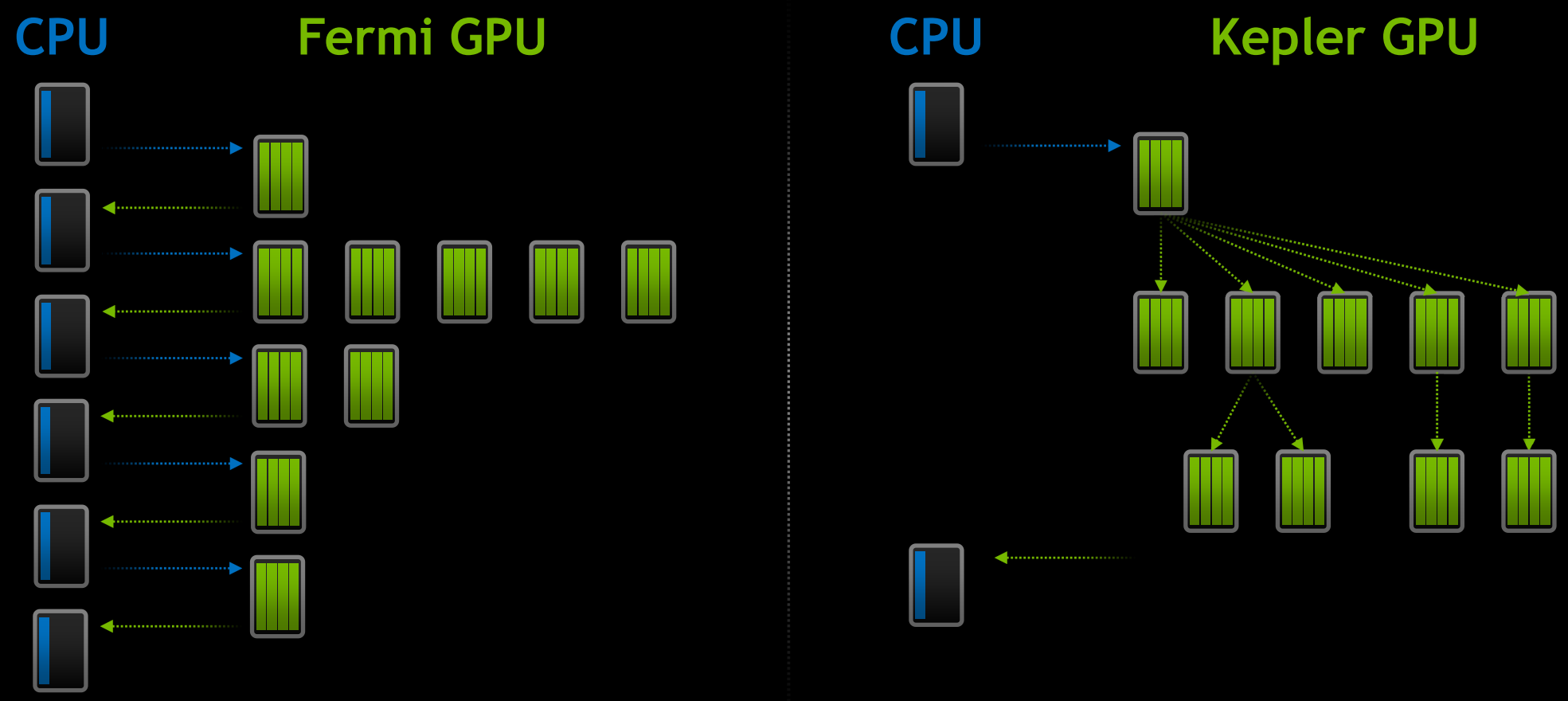
Strong Scaling: 864 water molecules

CP2K: MPI strong scaling

- Single-rank MPI fails to fully utilize GPU
- With Hyper-Q:
  - Code maintains one rank per CPU core (no need to refactor)
  - Dynamically share GPU
  - 2.5x speed-up

NVIDIA Confidential

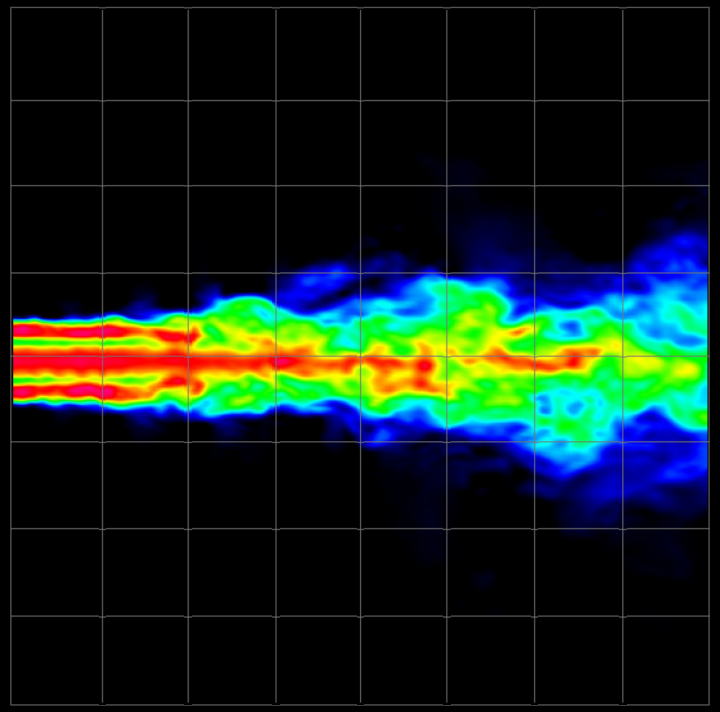
Dynamic Parallelism



NVIDIA Confidential

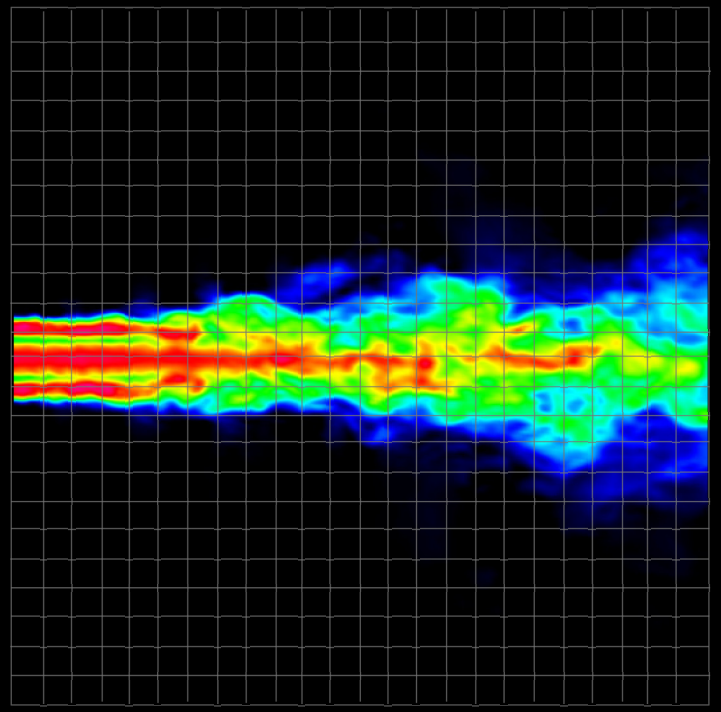
Dynamic Work Generation

Coarse grid



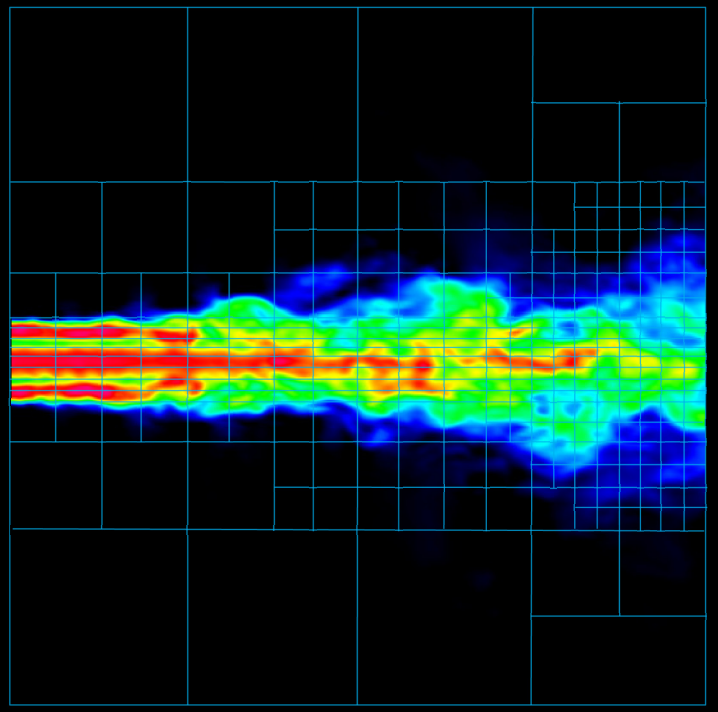
Higher Performance
Lower Accuracy

Fine grid



Higher Accuracy
Lower Performance

Dynamic grid



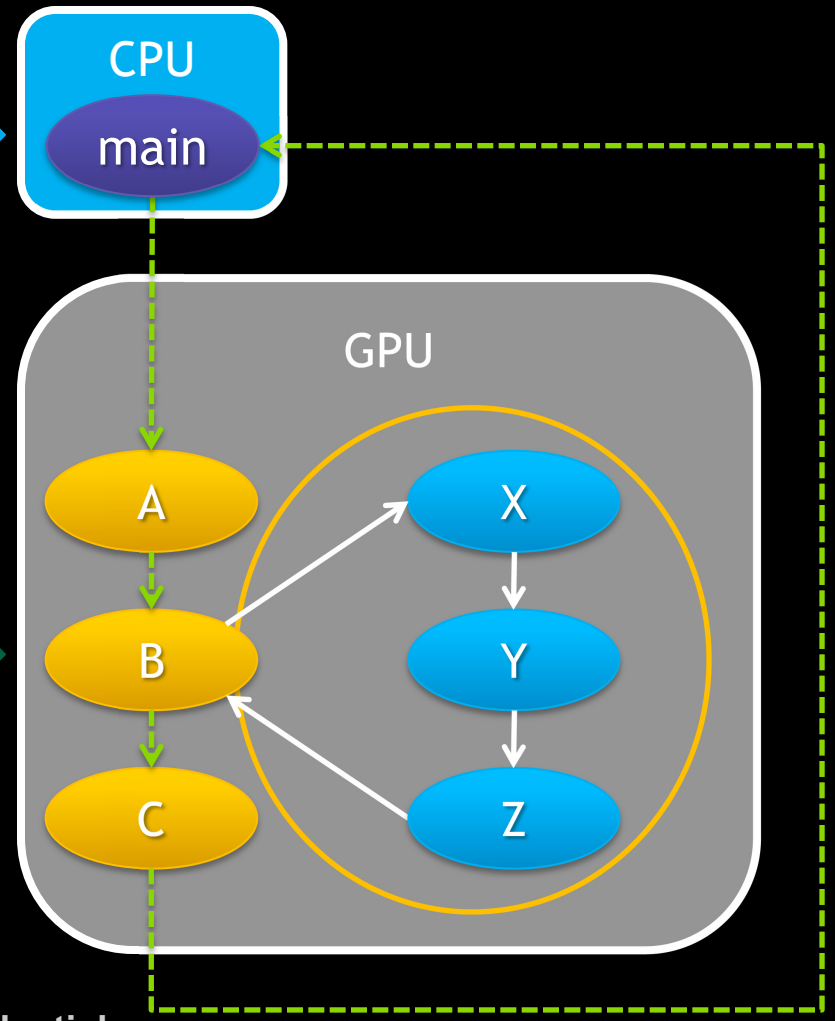
Target performance where accuracy is required

NVIDIA Confidential

Familiar Syntax and Programming Model

```
int main() {
    float *data;
    setup(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    return 0;
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```



NVIDIA Confidential

Simpler Code: LU Example

LU decomposition (Fermi)

```
dgetrf(N, N) {
  for j=1 to N
    for i=1 to 64
      idamax<<<>>>
      memcpy
      dswap<<<>>>
      memcpy
      dscal<<<>>>
      dger<<<>>>
    next i

    memcpy
    dlaswap<<<>>>
    dtrsm<<<>>>
    dgemm<<<>>>
  next j
}
```

```
idamax();
dswap();
dscal();
dger();
dlaswap();
dtrsm();
dgemm();
```

CPU Code | GPU Code

LU decomposition (Kepler)

```
dgetrf(N, N) {
  dgetrf<<<>>>
```

CPU is Free

```
  synchronize();
}
```

```
dgetrf(N, N) {
  for j=1 to N
    for i=1 to 64
      idamax<<<>>>
      dswap<<<>>>
      dscal<<<>>>
      dger<<<>>>
    next i
    dlaswap<<<>>>
    dtrsm<<<>>>
    dgemm<<<>>>
  next j
}
```

CPU Code | GPU Code

NVIDIA Confidential

Dynamic Parallelism: Simpler Code, Higher Perf

```
__device__ WorkStack stack;
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, push it on the stack
    if(left < nright)
        stack.push(data, left, nright);
    if(nleft < right)
        stack.push(data, nleft, right);
}

__host__ void launch_quicksort(int *data, int count)
{
    // Launch initial quicksort to populate the stack
    quicksort<<< ... >>>(data, 0, count-1);

    // Loop more quicksorts until no more work exists
    while(1)
    {
        // Wait for all sorts at this
        cudaDeviceSynchronize();

        // Copy our stack from the dev
        WorkStack stack_copy;
        stack_copy = CopyFromDevice(st

        // Count of things on stack. W
        if(stack_copy.size() == 0)
            break;

        // Pop the stack and launch ea
        while(stack_copy.size())
        {
            WorkStack elem = stack_cop
            cudaStream_t s;
            cudaStreamCreate(&s);
            quicksort<<< ..., s >>>(da
        }
    }
}
```

Without Dynamic Parallelism

```
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;
    cudaStream_t s1, s2;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, launch a new grid for it.
    // Note use of streams to get concurrency between sub-sorts
    if(left < nright) {
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        quicksort<<< ..., s1 >>>(data, left, nright);
    }
    if(nleft < right) {
        cudaStreamCreateWithFlags(&s2, cudaStreamNonBlocking);
        quicksort<<< ..., s2 >>>(data, nleft, right);
    }
}

__host__ void launch_quicksort(int *data, int count)
{
    quicksort<<< ... >>>(data, 0, count-1);
}
```

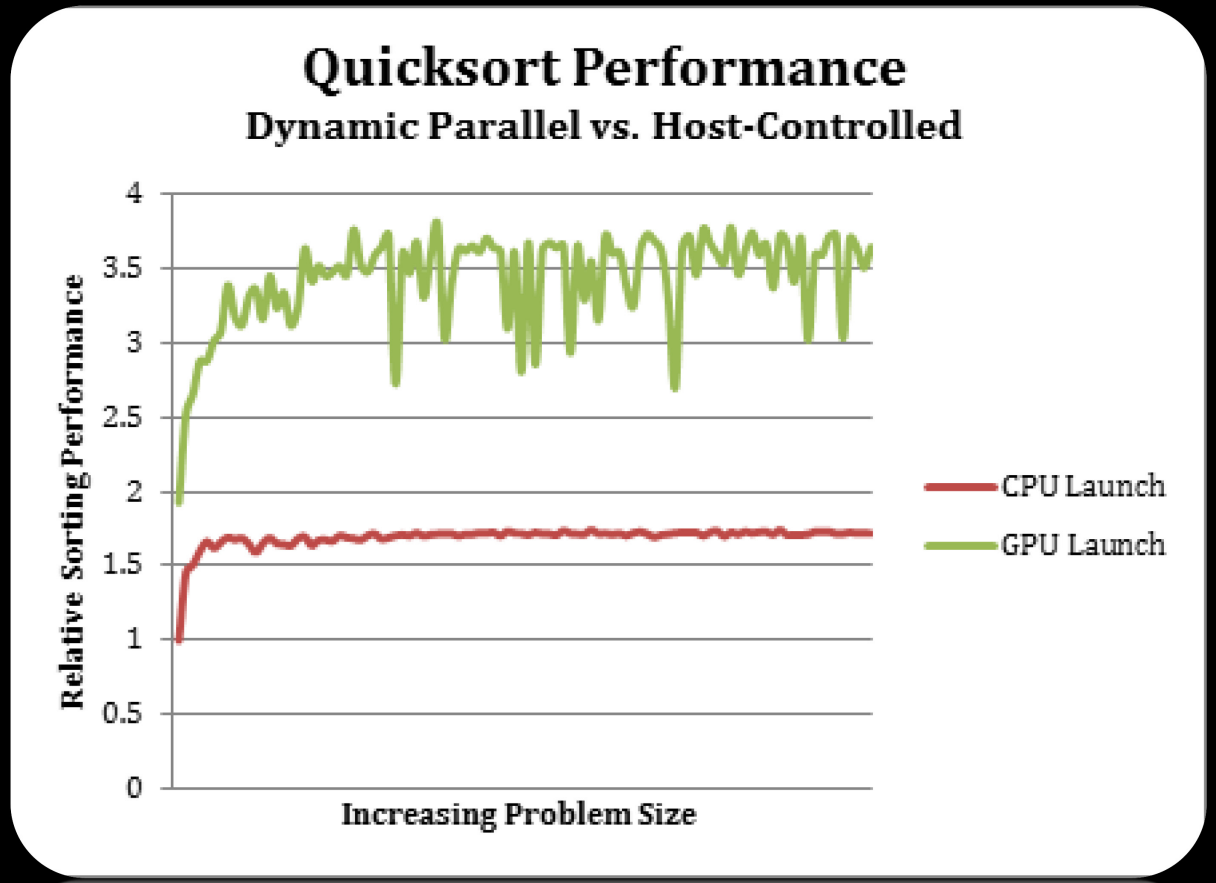
With Dynamic Parallelism

Quicksort: Parallel Recursion

- Code is easier & more natural, in <u>half the size</u>
- No complex overheads like stack exchange for recursion
- No complex CPU & GPU interaction
- Code looks like CPU code

NVIDIA Confidential

Dynamic Parallelism: Simpler Code, Higher Perf



Code Size Cut by 2x

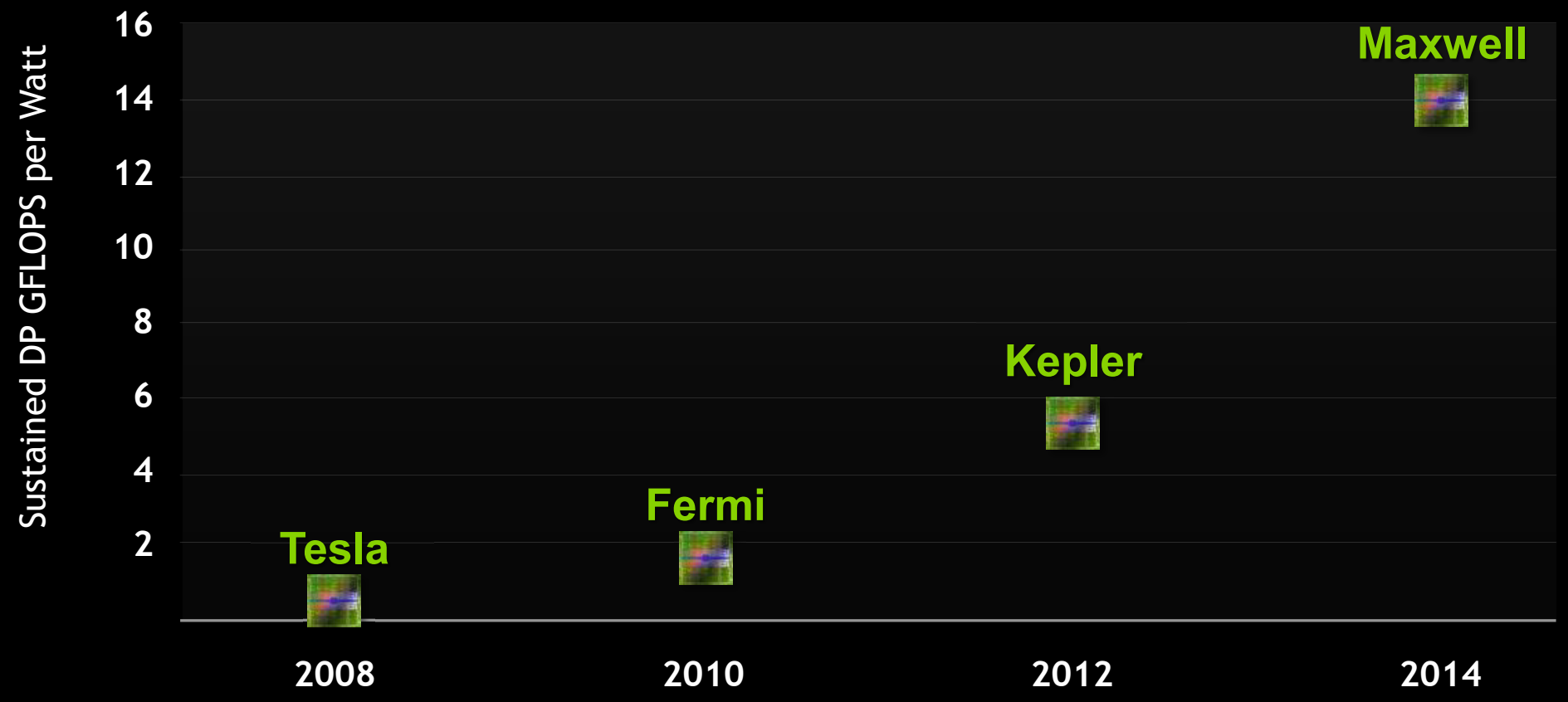
2x Performance

NVIDIA Confidential

Beyond Kepler…

NVIDIA Confidential

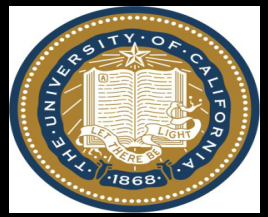
NVIDIA GPU Roadmap: Increasing Performance/Watt



NVIDIA Confidential

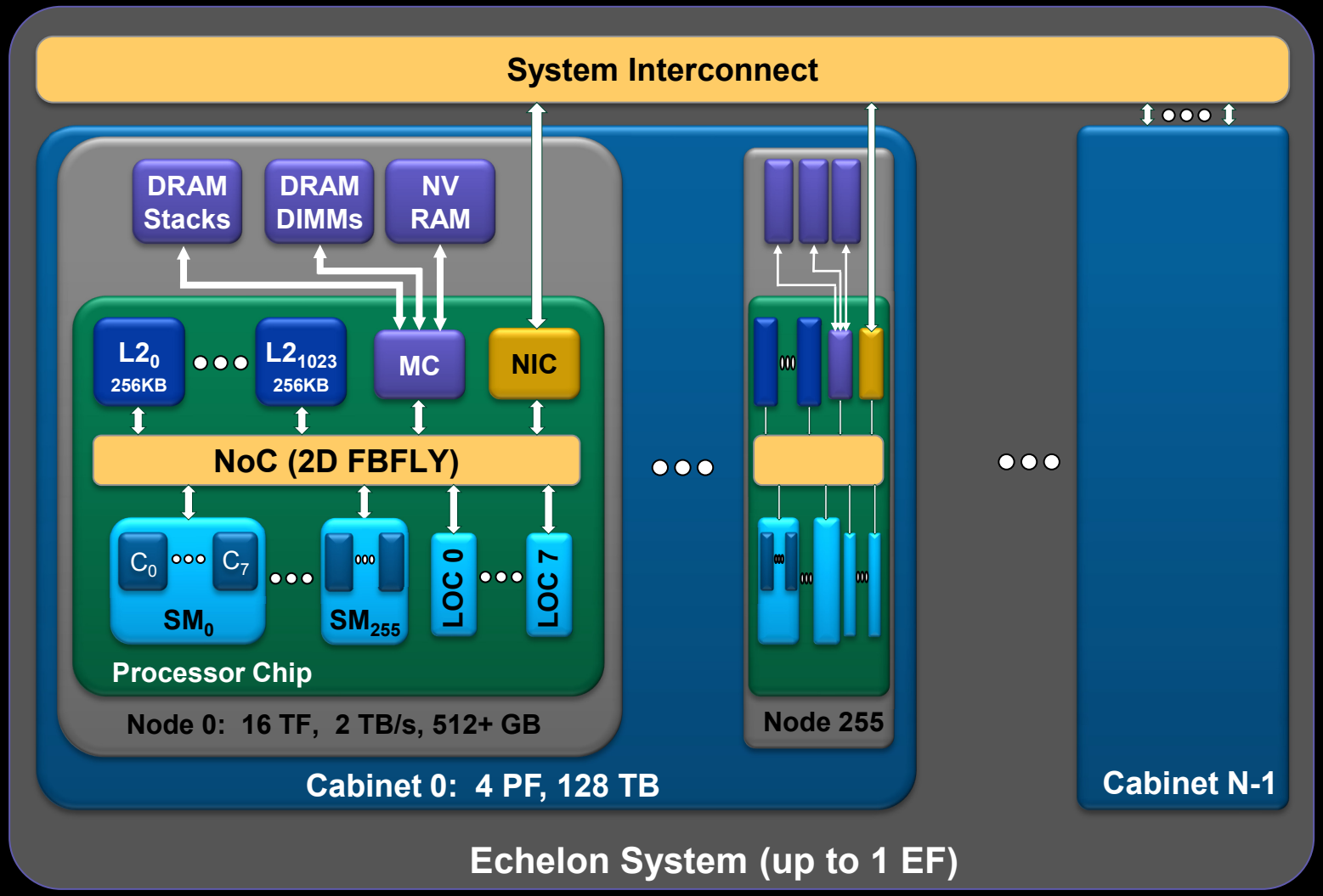
Echelon

NVIDIA's Extreme-Scale Computing Project
DARPA UHPC Program
Targeting 2018



NVIDIA Confidential

Vision for Exascale-Capable Processor in 2018



Key architectural features:

- Malleable memory hierarchy
- Hierarchical register files
- Hierarchical thread scheduling
- Place coherency/consistency
- Temporal SIMT & scalarization
- PGAS memory
- HW accelerated queues
- Active messages
- AMOs everywhere
- Collective engines
- Streamlined LOC/TOC interaction

Future of HPC Programming

Computers are not getting faster… just **wider**

⇒ *Need to cast all HPC apps as throughput problems, and expose massive parallelism*

Locality across nodes is not the problem… *vertical* locality is

⇒ *Need to expose locality & explicitly manage memory hierarchy*

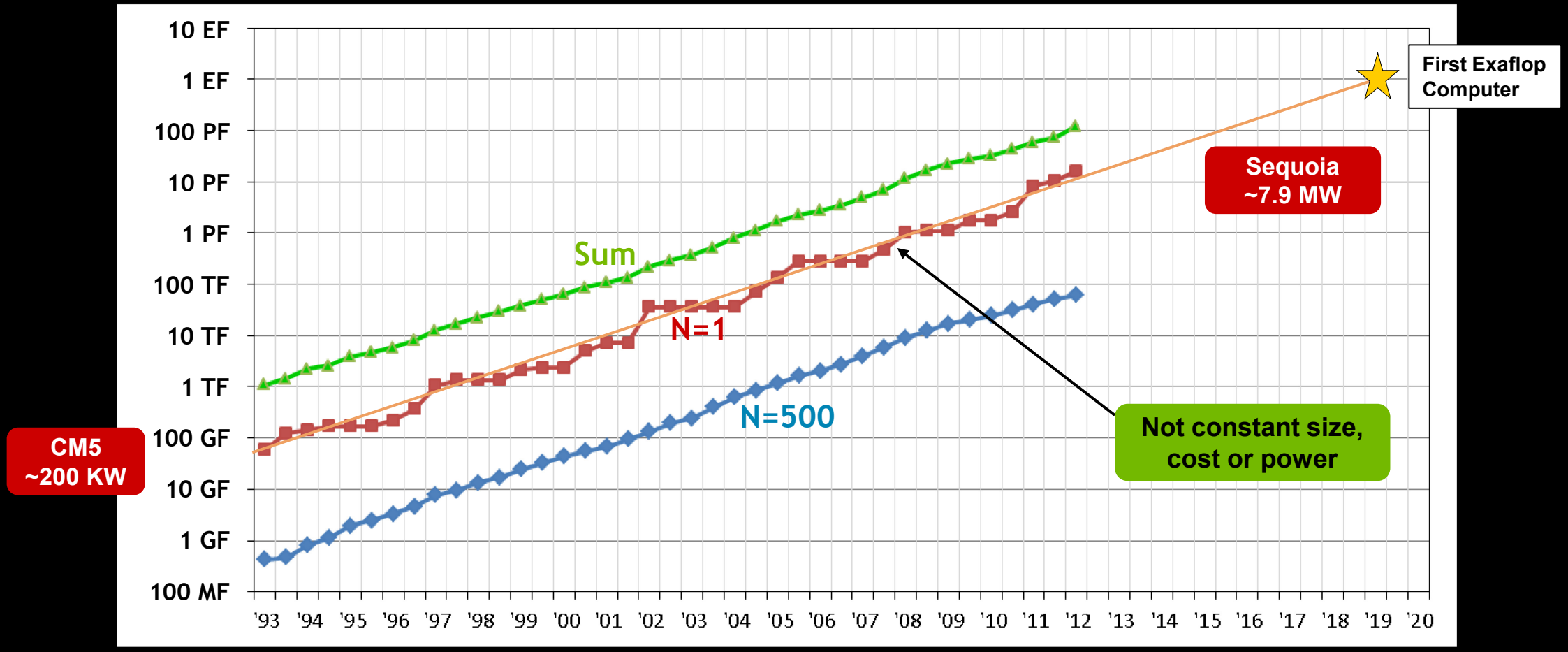
(programming model) *(compiler, runtime, auto-tuner)*

Will need to restructure codes for the future,
but want to maintain (spatial and temporal) portability

NVIDIA Confidential

Thank You.

Exaflop Expectations



Power: This Time It's Different

In the Good Old Days
Leakage was not important, and voltage scaled with feature size

$L' = L/2$
$V' = V/2$
$E' = CV^2 = E/8$
$f' = 2f$
$D' = 1/L^2 = 4D$
$P' = P$

Halve L and get 4x the transistors and 8x the capability for the same power!

MF to GF to TF and almost to PF

Technology was giving us **68%** per year in perf/W!

Processors realized ~50% per year in perf/W…
(spent it on single thread performance)

The New Reality
Leakage has limited threshold voltage, largely ending voltage scaling

Halve L and get 2x the capability for the same power.

At constant voltage, technology gives us only **19%** per year in perf/W

NVIDIA Confidential