
Heterogeneous
Computing
David Luebke
NVIDIA

The “New” Moore’s Law

- Computers no longer get faster, just wider
- You *must* re-think your algorithms to be parallel !
- Data-parallel computing is most scalable solution

Enter the GPU

- Massive economies of scale
- Massively parallel

Enter CUDA

- Scalable parallel programming model
- Minimal extensions to familiar C/C++ environment
- Heterogeneous serial-parallel computing

Sound Bite

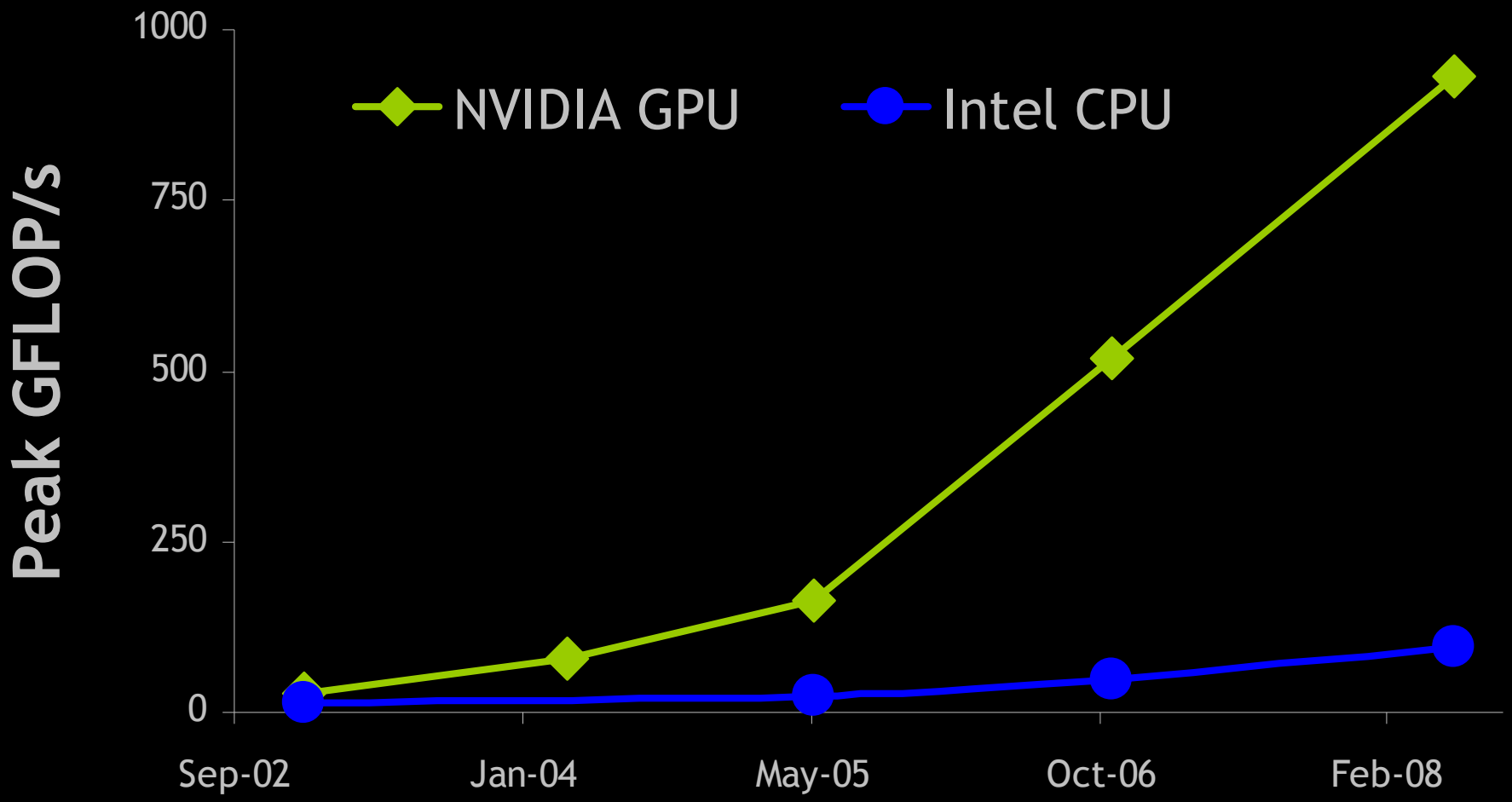
GPUs + CUDA

=

The Democratization of Parallel Computing

Massively parallel computing has become a commodity technology

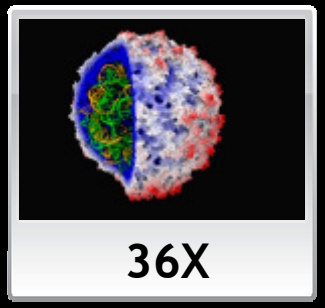
MOTIVATION


Peak GFLOP/s
1000
750
500
250
0
NVIDIA GPU
Intel CPU
Sep-02
Jan-04
May-05
Oct-06
Feb-08

MOTIVATION

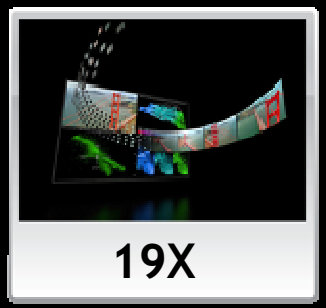
146X

Interactive visualization of volumetric white matter connectivity



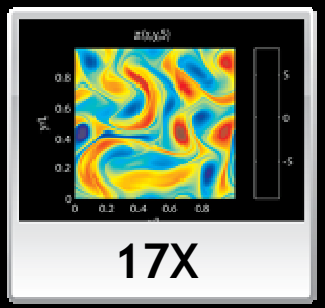
36X

Ionic placement for molecular dynamics simulation on GPU



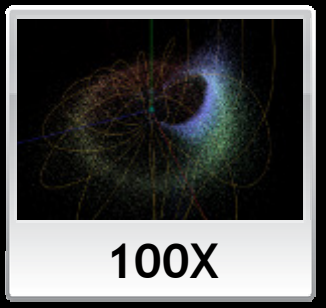
19X

Transcoding HD video stream to H.264



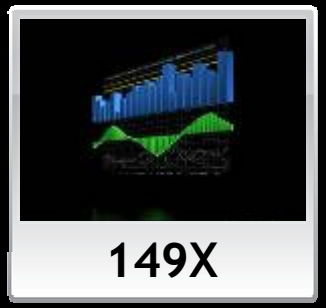
17X

Fluid mechanics in Matlab using .mex file CUDA function



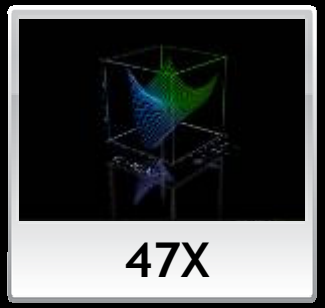
100X

Astrophysics N-body simulation



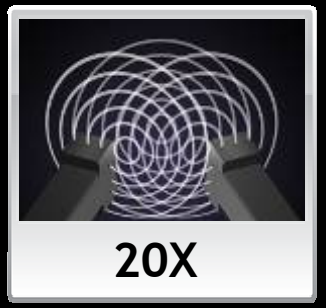
149X

Financial simulation of LIBOR model with swaptions



47X

GLAME@lab: an M-script API for GPU linear algebra

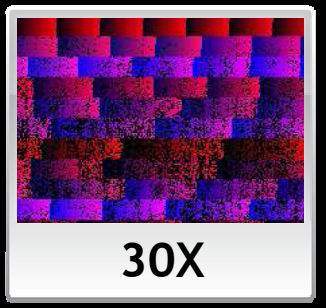


20X

Ultrasound medical imaging for cancer diagnostics

24X

Highly optimized object oriented molecular dynamics



30X

Cmatch exact string matching to find similar proteins and gene sequences

Motivation: NVIDIA

Supercomputing Performance

- 960 cores. 4 TeraFLOPS
- 250x the performance of a desktop

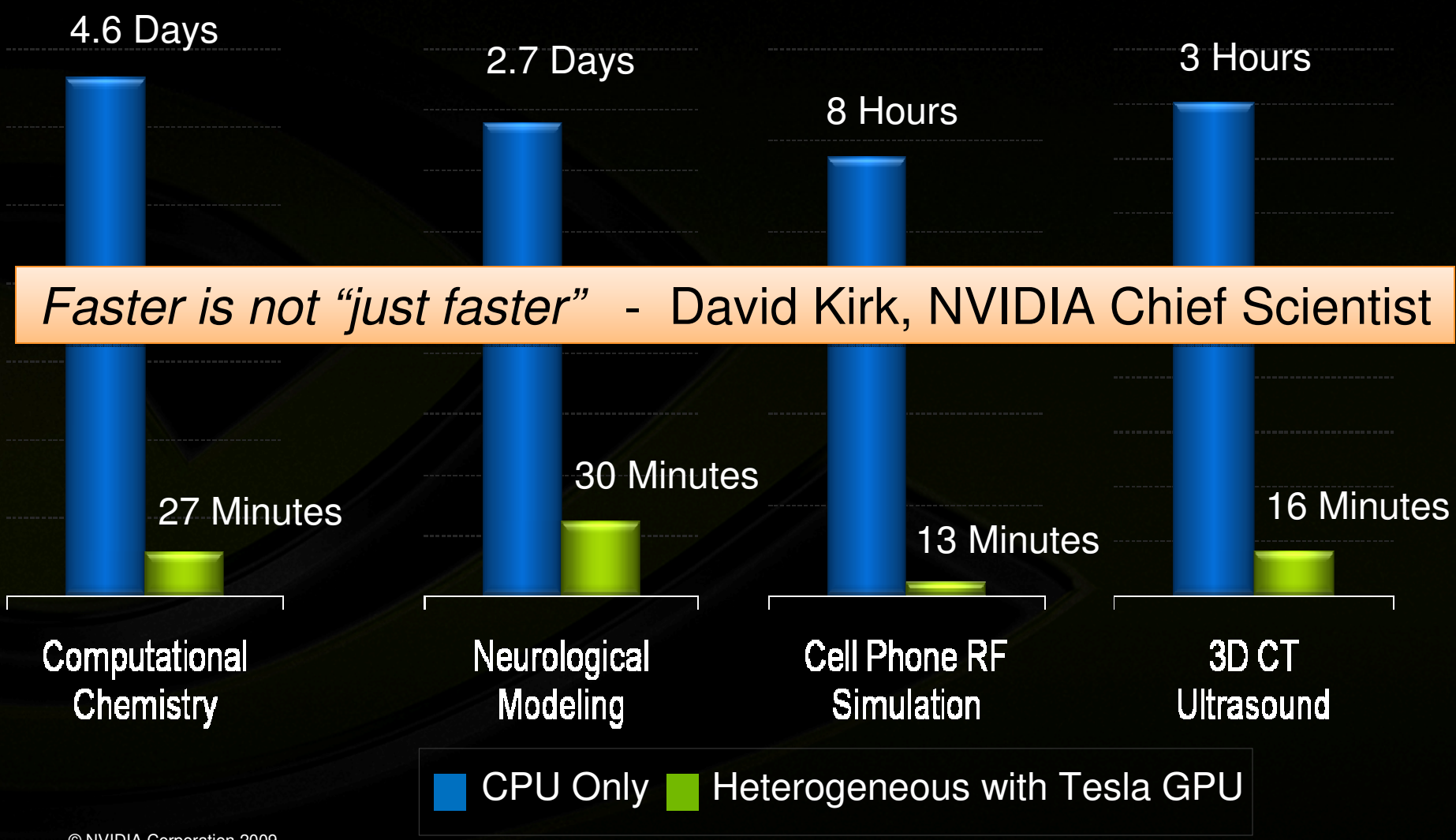
Personal

- One researcher, one supercomputer
- Plugs into standard power strip

Accessible

- Program in C for Windows, Linux
- Available now under $10,000

Accelerating Time to Insight



Faster is not "just faster" - David Kirk, NVIDIA Chief Scientist

CUDA: 'C' FOR PARALLELISM


```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

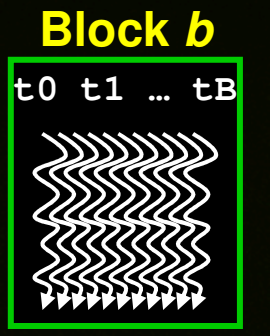
```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

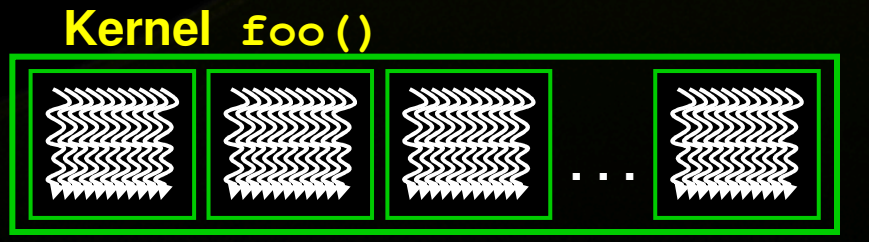
Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program

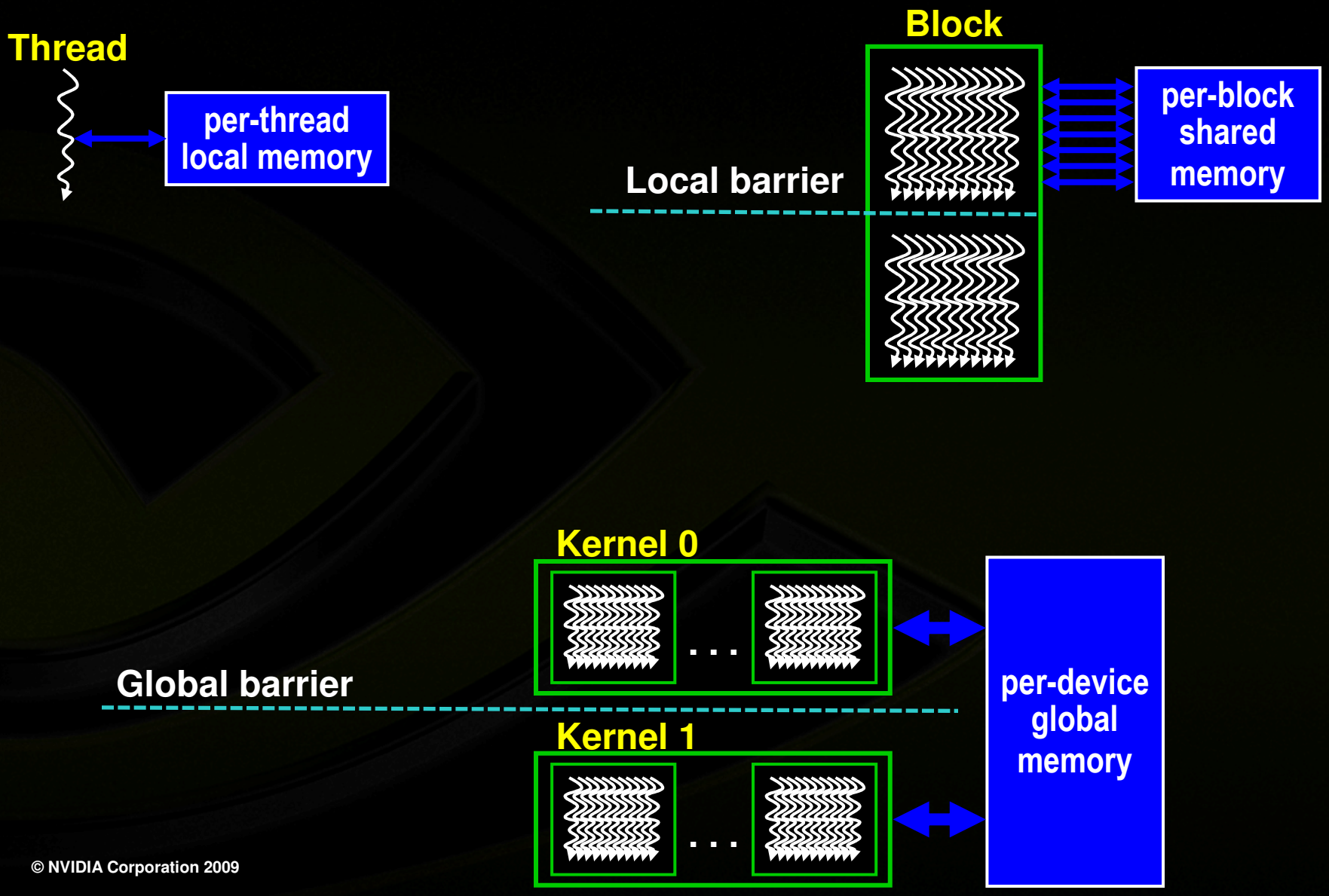
- Threads are grouped into thread blocks
  - threads in the same block can cooperate



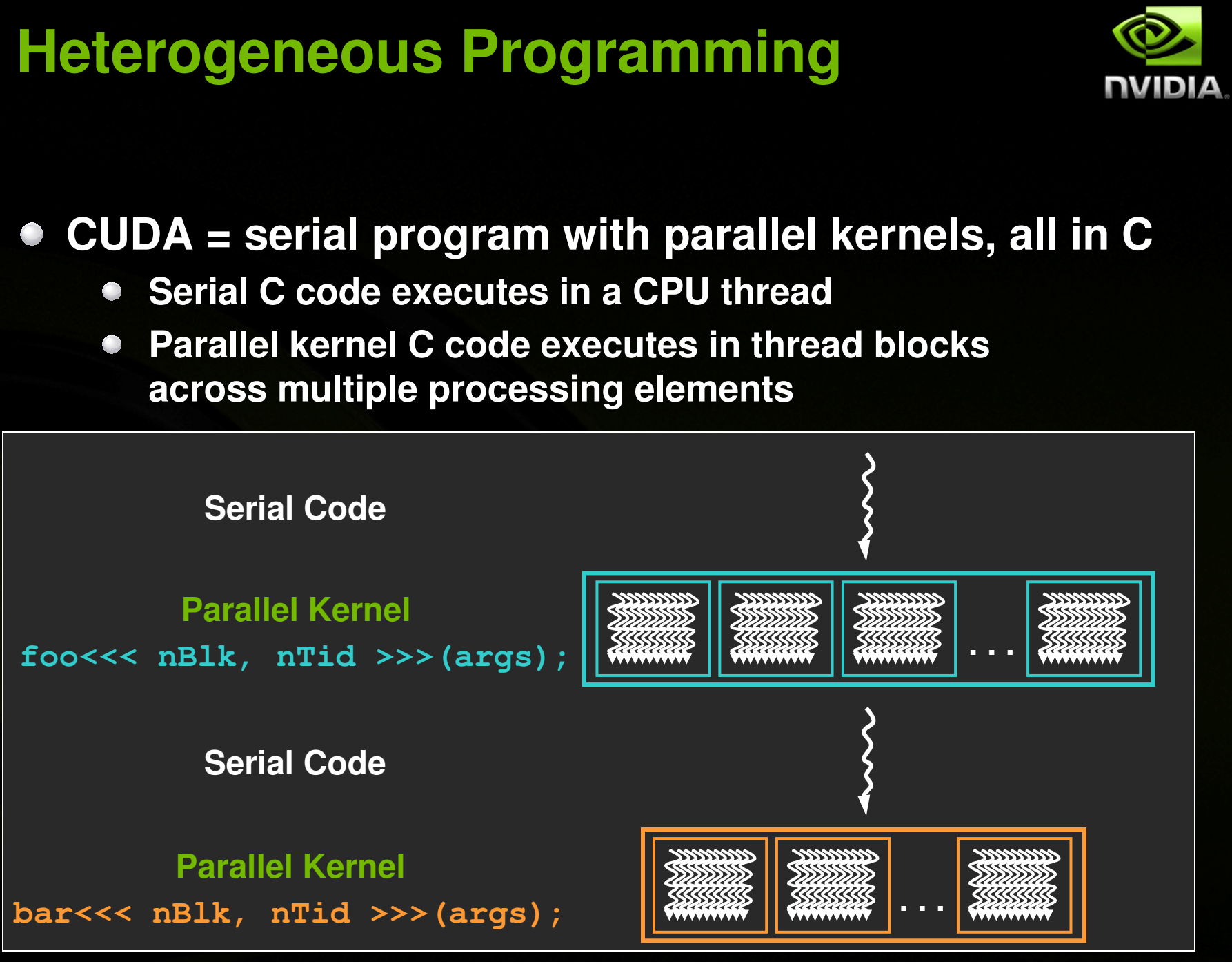
- Threads/blocks have unique IDs



Hierarchical organization



Heterogeneous Programming



- CUDA = serial program with parallel kernels, all in C
  - Serial C code executes in a CPU thread
  - Parallel kernel C code executes in thread blocks across multiple processing elements

Serial Code

Parallel Kernel

```
foo<<< nBlk, nTid >>>(args);
```

Serial Code

Parallel Kernel

```
bar<<< nBlk, nTid >>>(args);
```

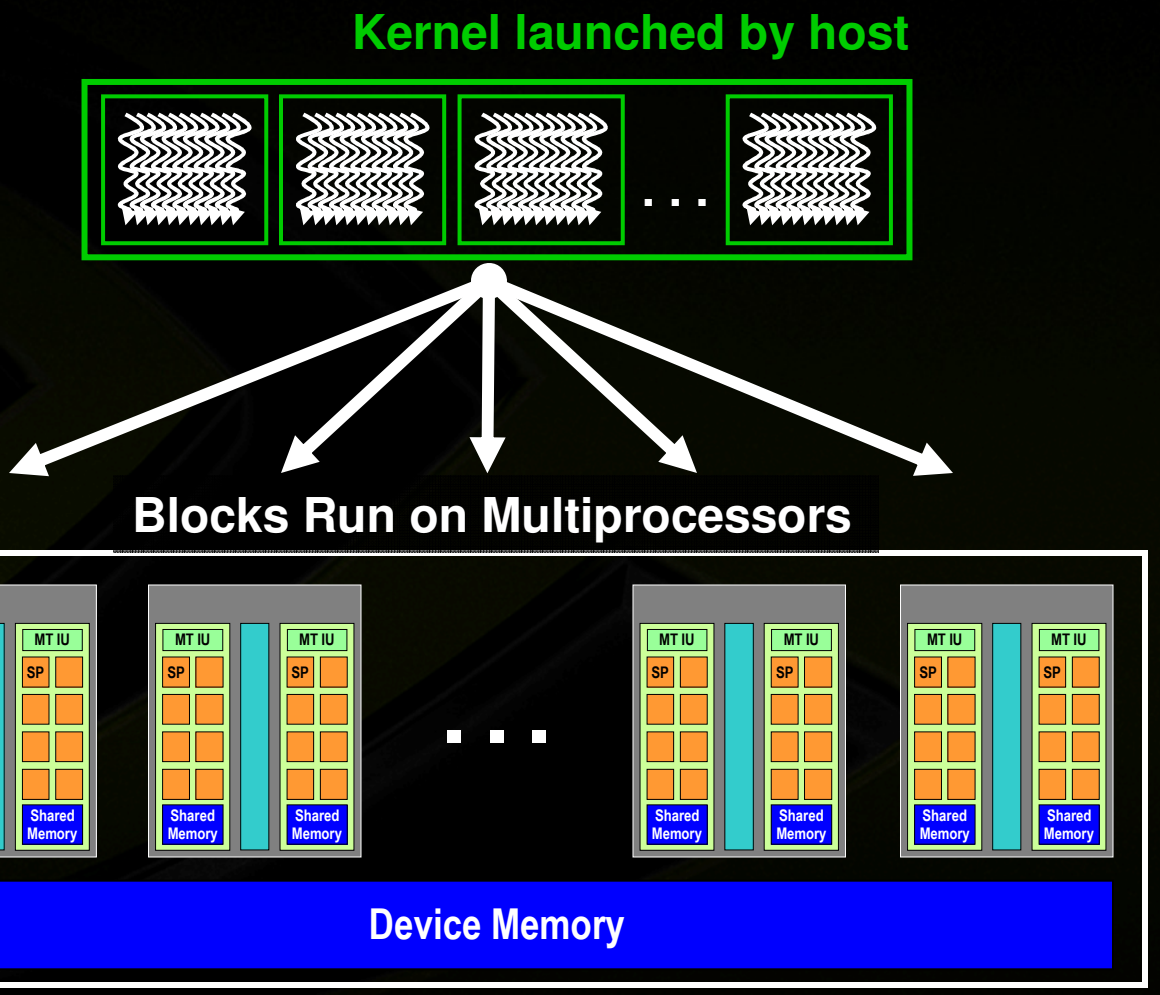
Thread = virtualized scalar processor

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

Block = virtualized multiprocessor

- Provides programmer flexibility
  - freely choose processors to fit data
  - freely customize for each kernel launch
- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want
- Thread blocks of kernel must be independent tasks
  - program valid for *any interleaving* of block executions

Scalable Execution Model



Synchronization & Cooperation

- Threads within block may synchronize with barriers

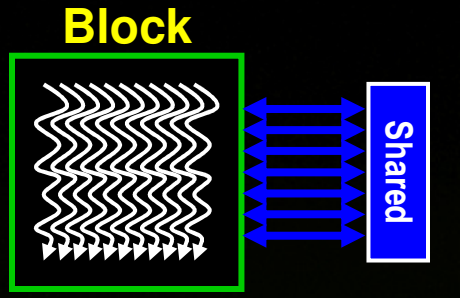
```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks coordinate via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between dependent kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```

Using per-block shared memory



- Variables shared across block

```
__shared__ int *begin, *end;
```

- Scratchpad memory

```
__shared__ int scratch[blocksize];
scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

Summing Up

- CUDA = C + a few simple extensions
  - makes it easy to start writing basic parallel programs
- Three key abstractions:
  1. hierarchy of parallel threads
  2. corresponding levels of synchronization
  3. corresponding memory spaces
- Supports massive parallelism of manycore GPUs

SOME FINAL THOUGHTS

- We should teach parallel computing in CS 1 or CS 2
- Remember: computers don't get faster, just wider
- Heapsort and mergesort
  - Both O(n lg n)
  - One parallel-friendly, one not
  - Students need to understand this early

Conclusion

- GPUs are massively parallel manycore computers
  - Ubiquitous - most successful parallel processor in history
  - Useful - users achieve huge speedups on real problems
- CUDA is a powerful parallel architecture and programming model
  - Heterogeneous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible – e.g. minimal but expressive changes to C
- They provide tremendous scope for innovative research


Questions?
NVIDIA

Example: Vector Add w/ Host Code

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel


```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Host code for vecAdd


```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
   cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
   cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

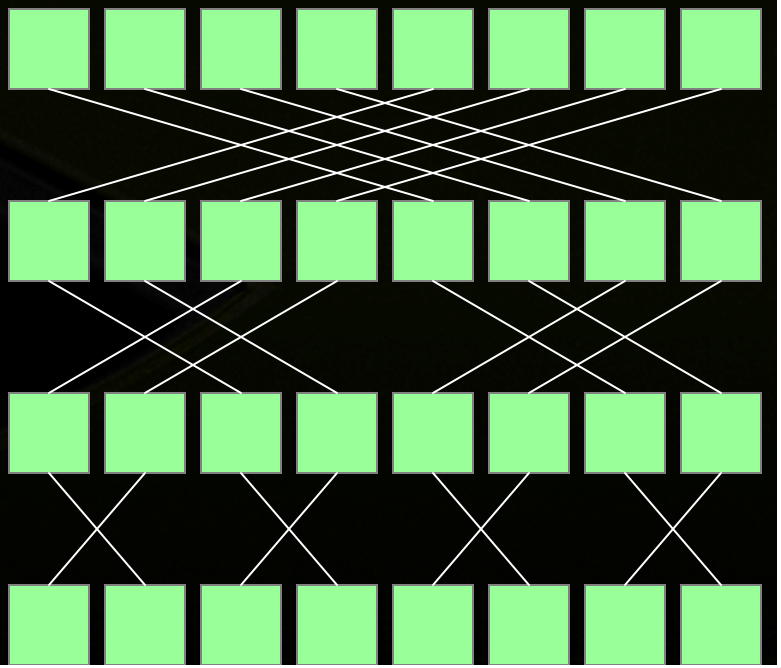
Example: Reduction

Example: Parallel Reduction

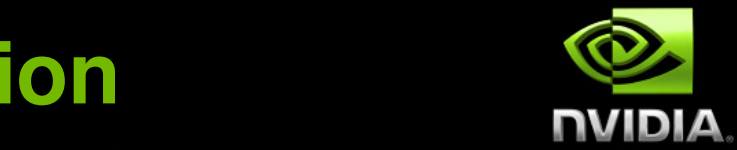
- Summing up a sequence with 1 thread:

```
int sum = 0;
for(int i=0; i<N; ++i)  sum += x[i];
```

- Parallel reduction builds a summation tree
  - each thread holds 1 element
  - stepwise partial sums
  - N threads need log N steps
  - one possible approach: Butterfly pattern



Example: Parallel Reduction

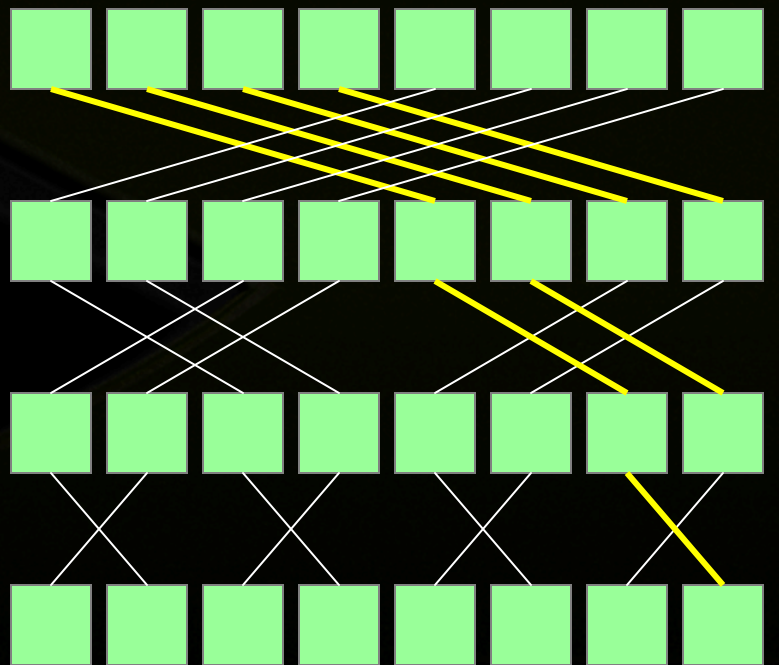


- Summing up a sequence with 1 thread:

```
int sum = 0;
for(int i=0; i<N; ++i)  sum += x[i];
```

- Parallel reduction builds a summation tree
  - each thread holds 1 element
  - stepwise partial sums
  - N threads need log N steps

  - one possible approach: Butterfly pattern

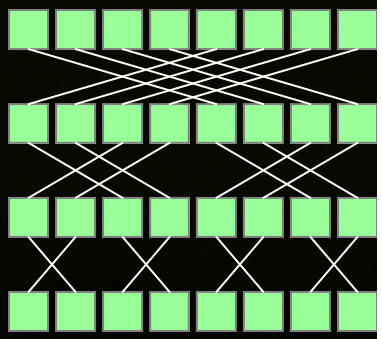


Parallel Reduction for 1 Block


```
// INPUT: Thread i holds value x_i
int i = threadIdx.x;
__shared__ int sum[blocksize];
```

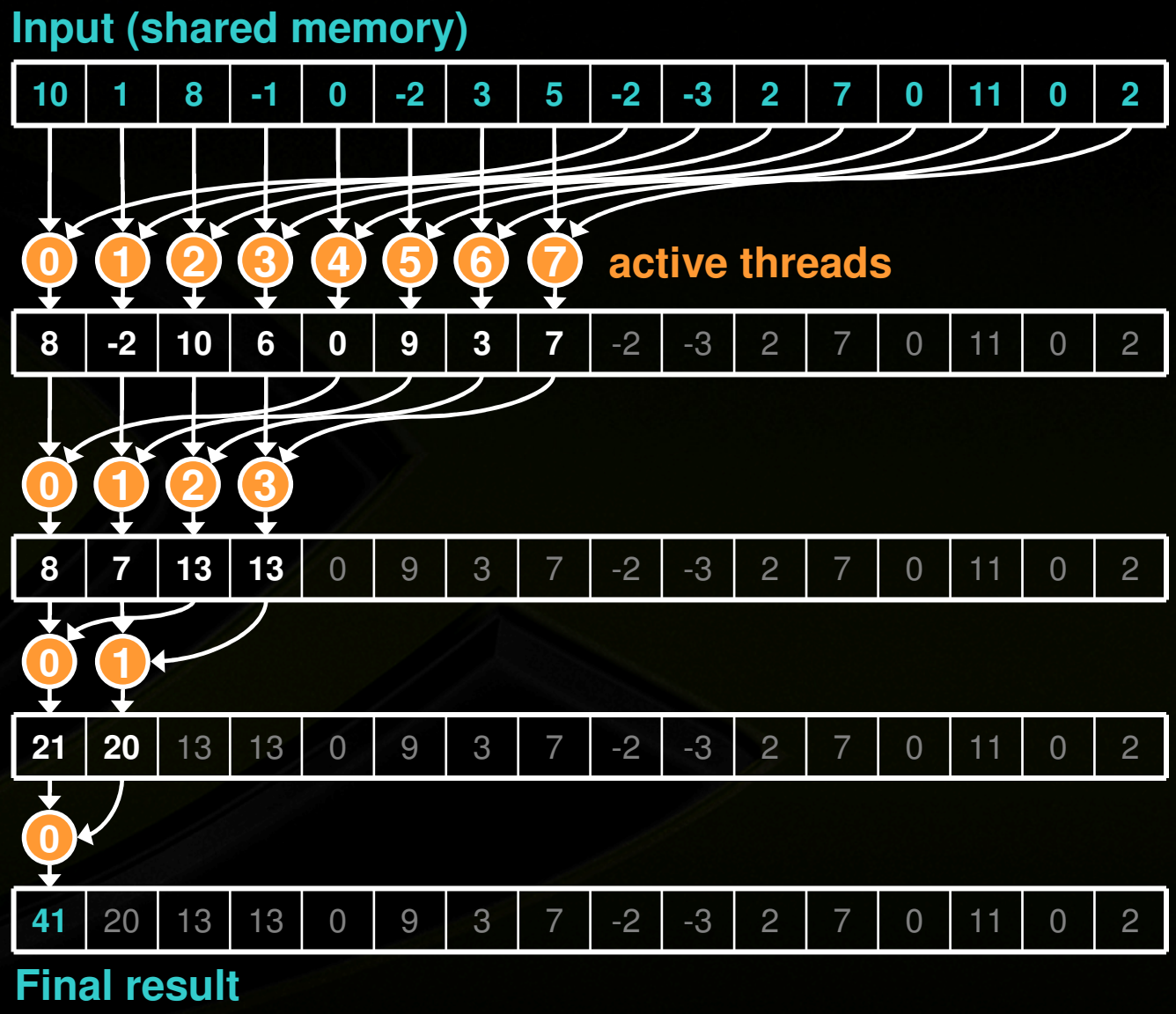
```
// One thread per element
sum[i] = x_i; __syncthreads();
```

```
for(int bit=blocksize/2; bit>0; bit/=2)
{
    int t=sum[i]+sum[i^bit]; __syncthreads();
    sum[i]=t;                __syncthreads();
}
```

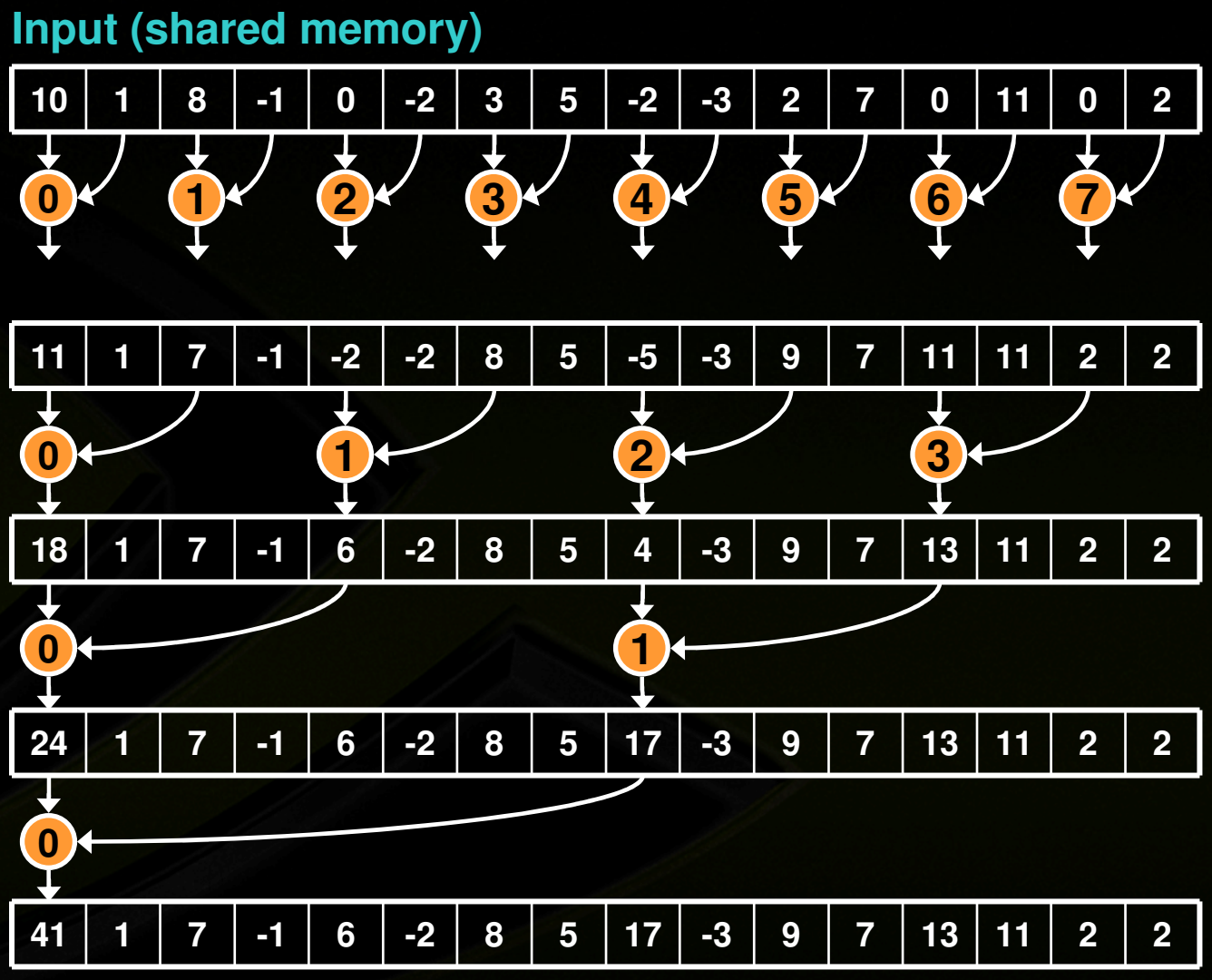


```
// OUTPUT: Every thread now holds sum in sum[i]
```

Reduction tree redux

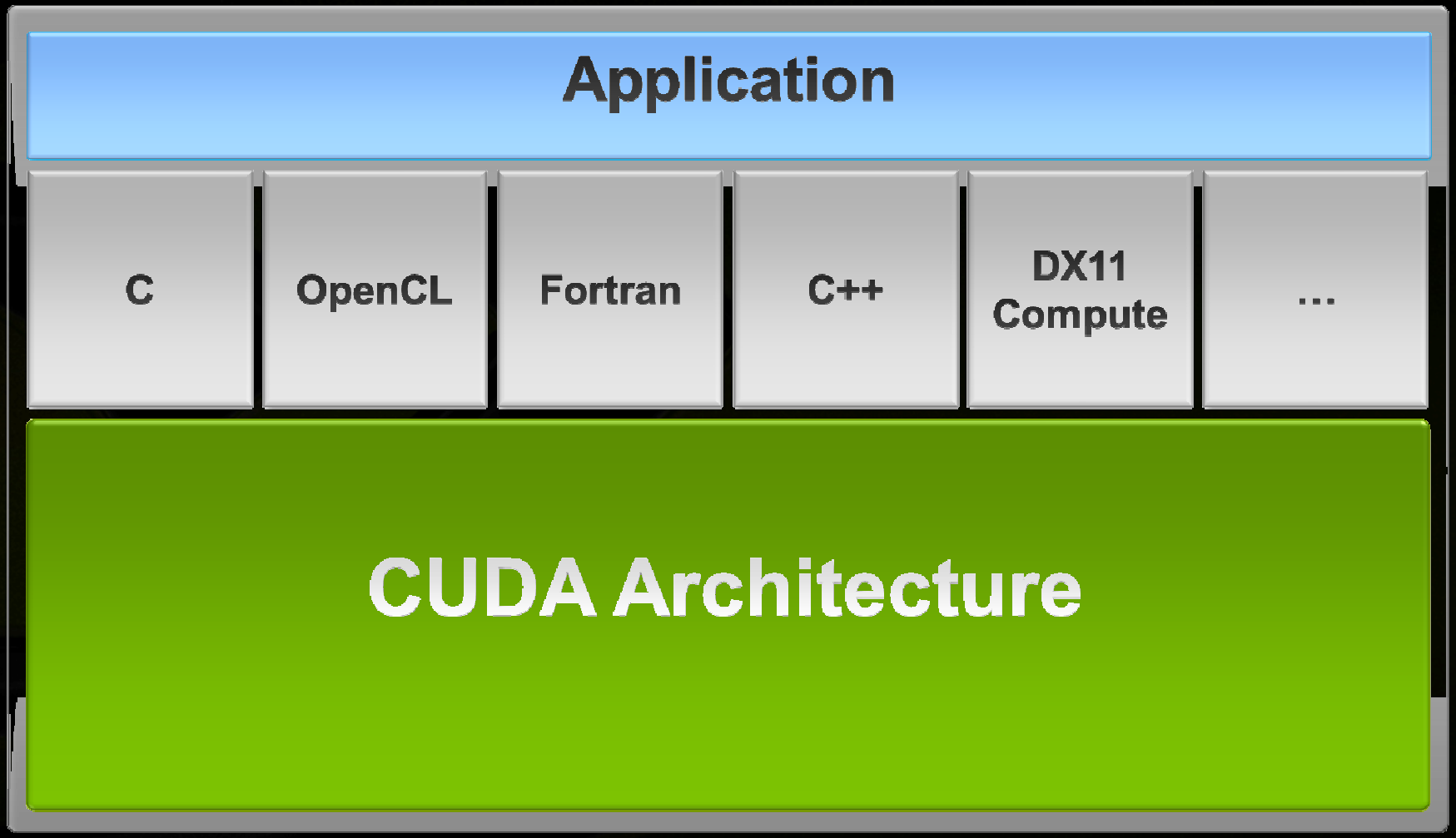


Compare to interleaved addressing:

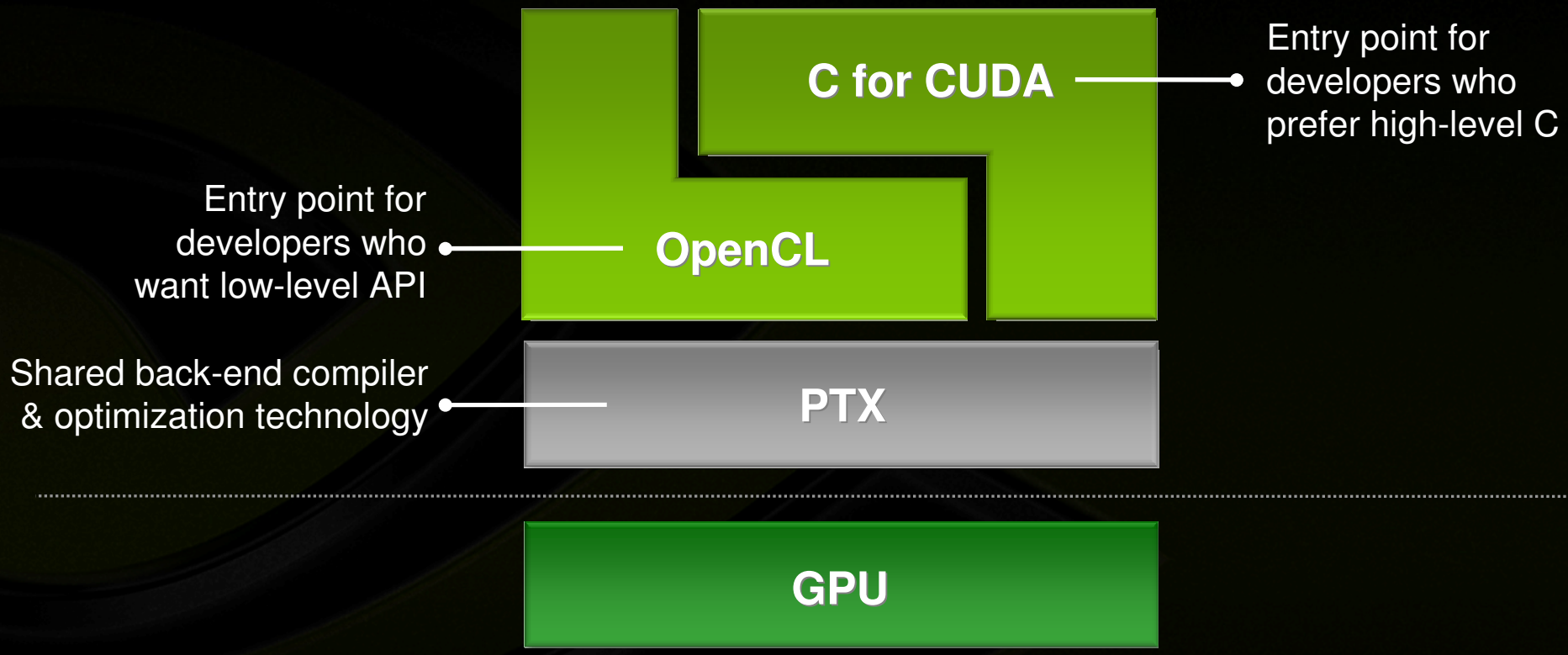


OpenCL

CUDA: An Architecture for Massively Parallel Computing



OpenCL vs. C for CUDA



FFT Kernel Example

OPENCL

```
__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy)
{
  int tid = get_local_id(0);   int blockIdx = get_group_id(0) * 1024 + tid;
  float2 data[16];
  in = in + blockIdx;  out = out + blockIdx;

  globalLoads(data, in, 64); // coalesced global reads
  fftRadix16Pass(data);      // in-place radix-16 pass
  twiddleFactorMul(data, tid, 1024, 0);
  localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
  fftRadix16Pass(data);              // in-place radix-16 pass
  twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
  localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));
  fftRadix4Pass(data);
  fftRadix4Pass(data + 4); // four radix-4 function calls
  fftRadix4Pass(data + 8)
  fftRadix4Pass(data + 12);
  globalStores(data, out, 64); // coalesced global writes
}
```

Calculate Index
Load Data

FFT Kernel

C for CUDA (Written by Vasily Volkov, © UC

```
__global__ void FFT1024_device( float2 *dst, float2 *src )
{
  int tid = threadIdx.x; int iblock = blockIdx.y * gridDim.x + blockIdx.x;
  int index = iblock * 1024 + tid; src += index; dst += index;
  int hi4 = tid>>4; int lo4 = tid&15;int hi2 = tid>>4; int mi2 = (tid>>2)&3;int
      lo2 = tid&3;
  float2 a[16];
  __shared__ float smem[69*16];

   load<16>( a, src, 64 );
  FFT16( a );
  twiddle<16>( a, tid, 1024 );
  int il[] = {0,1,2,3, 16,17,18,19, 32,33,34,35, 48,49,50,51};
  transpose<16>( a, &smem[lo4*65+hi4], 4, &smem[lo4*65+hi4*4], il );
  FFT4x4( a );
  twiddle4x4( a, lo4 );
  transpose4x4( a, &smem[hi2*17 + mi2*4 + lo2], 69, &smem[mi2*69*4 +
      hi2*69 + lo2*17 ], 1, 0xE );
  FFT16( a );
  store<16>( a, dst, 64 );
}
```

Different Host Code Styles

Calling a C function in nvcc

```
extern "C" void FFT1024( float2 *work, int batch )
{
    FFT1024_device<<< grid2D(batch), 64 >>>
      ( work, work );
}
```

NVIDIA's PTX layer manages kernel resources and execution

OpenCL API-style programming

```
// create a compute context with GPU device
context = clCreateContextFromType(CL_DEVICE_TYPE_GPU);
// create a work-queue
queue = clCreateWorkQueue(context, NULL, NULL, 0);
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(float)*2*num_entries, srcA);
memobjs[1] = clCreateBuffer(context,
CL_MEM_READ_WRITE,
sizeof(float)*2*num_entries, NULL);
// create the compute program
program = clCreateProgramFromSource(context, 1,
&fft1D_1024_kernel_src, NULL);
// build the compute program executable
clBuildProgramExecutable(program, false, NULL, NULL);
// create the compute kernel
kernel = clCreateKernel(program, "fft1D_1024");
// create N-D range object with work-item dimensions
global_work_size[0] = n;
local_work_size[0] = 64;
range = clCreateNDRangeContainer(context, 0, 1,
global_work_size,
local_work_size);
// set the args values
clSetKernelArg(kernel, 0, (void *)&memobjs[0],
sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 1, (void *)&memobjs[1],
sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 2, NULL,
sizeof(float)*(local_work_size[0]+1)*16, NULL);
clSetKernelArg(kernel, 3, NULL,
sizeof(float)*(local_work_size[0]+1)*16, NULL);
// execute kernel
clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```

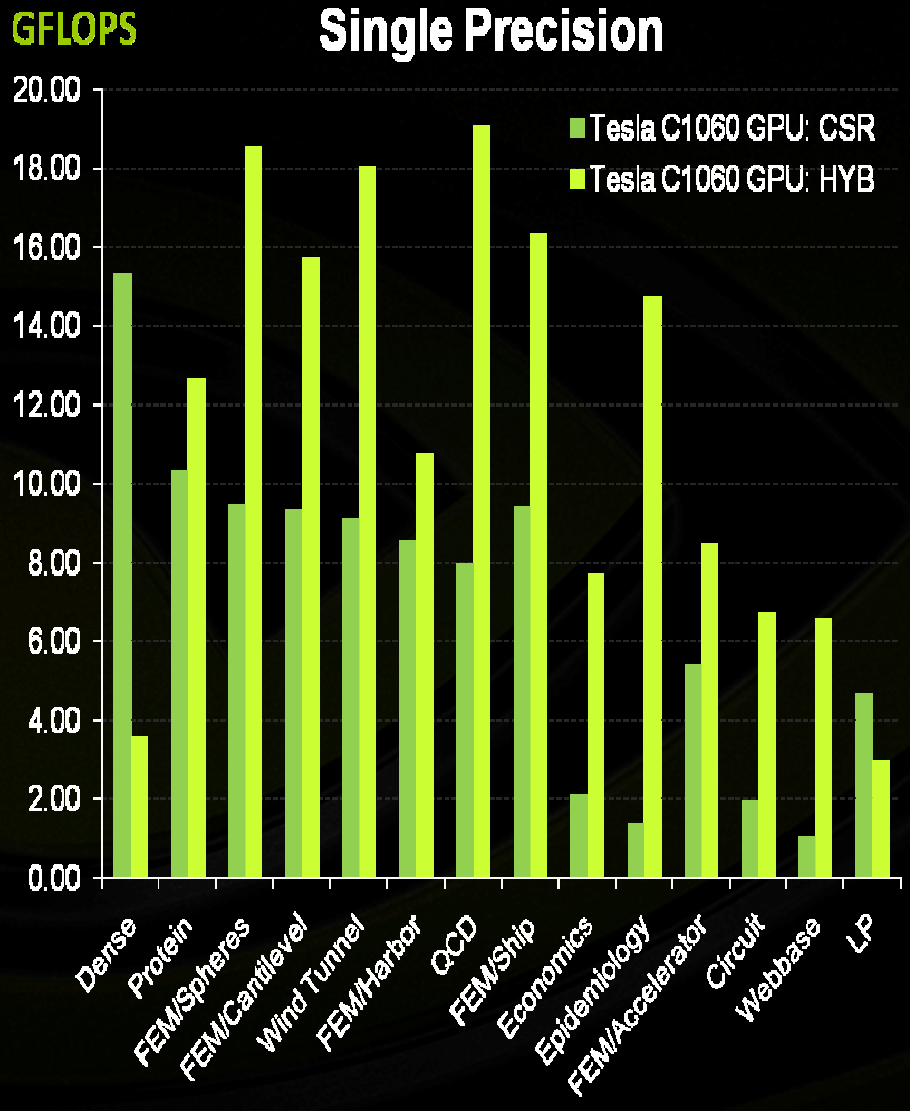
Source:
SIGGraph sneak preview
A Munshi, Apple Computer

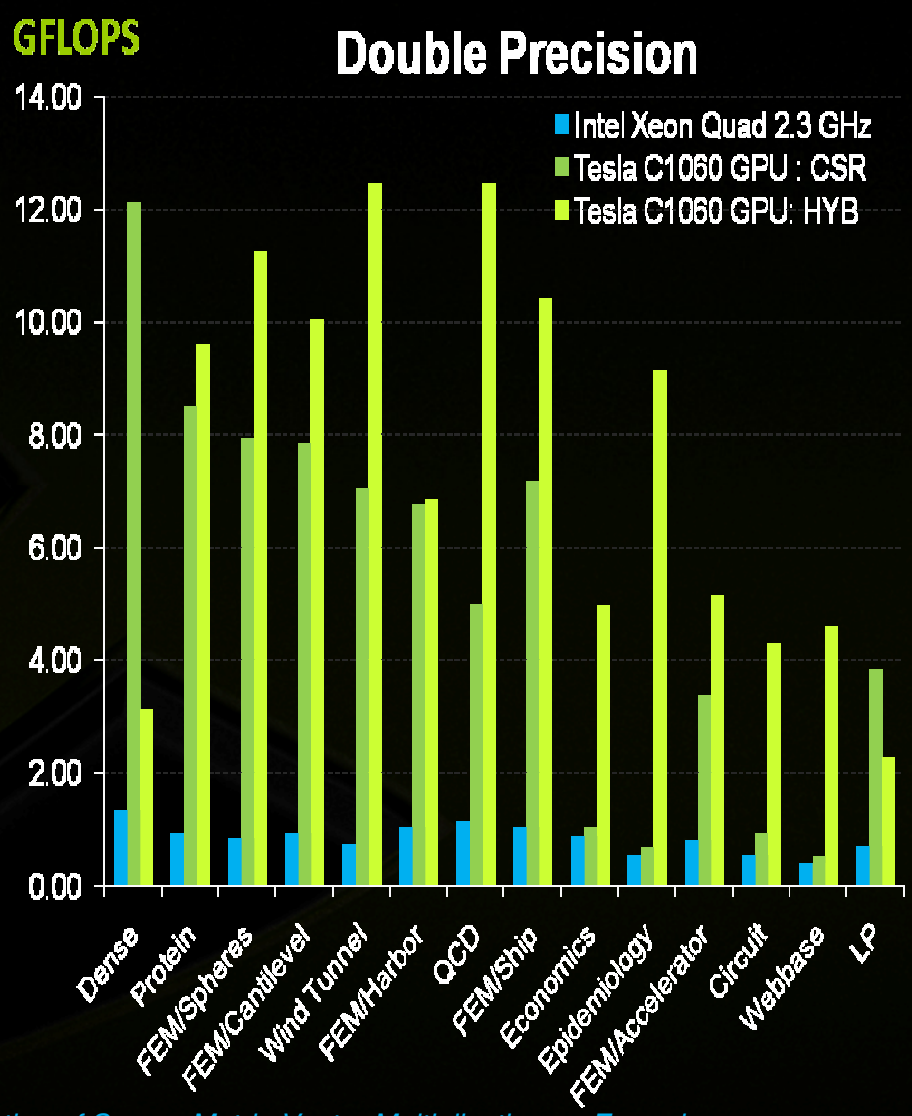
Sparse Linear Algebra Results

Sparse Matrix-Vector Multiplication (SpMV) on CUDA

- Experimented with several data structures
  - CSR: Compressed Sparse Row
  - HYB: Hybrid of ELLPACK (ELL) and Coordinate (COO) formats
- HYB gave best results
  - Speed of ELL with flexibility of COO
- Benchmarked against matrices from
  - *"Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms"*, S. Williams *et al*, Supercomputing 2007

Results: Sparse Matrix-Vector Multiplication (SpMV) on CUDA





CPU Results from "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms", Williams et al, Supercomputing 2007

Double Precision

T10 Double Precision Floating Point

Precision	IEEE 754
Rounding modes for FADD and FMUL	All 4 IEEE, round to nearest, zero, inf, -inf
Denormal handling	Full speed
NaN support	Yes
Overflow and Infinity support	Yes
Flags	No
FMA	Yes
Square root	Software with low-latency FMA-based convergence
Division	Software with low-latency FMA-based convergence
Reciprocal estimate accuracy	24 bit
Reciprocal sqrt estimate accuracy	23 bit
log2(x) and 2^x estimates accuracy	23 bit

Double Precision Floating Point

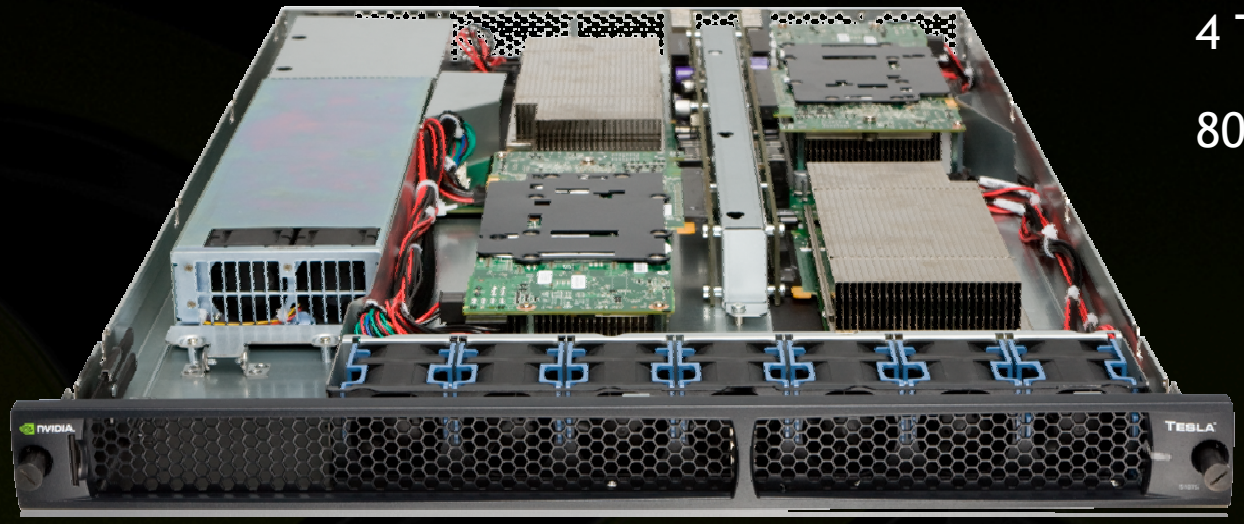
	NVIDIA Tesla T10	SSE2	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	All 4 IEEE, round to nearest, zero, inf, -inf	All 4 IEEE, round to nearest, zero, inf, -inf	All 4 IEEE, round to nearest, zero, inf, -inf
Denormal handling	Full speed	Supported, costs 1000's of cycles	Supported only for results, not input operands (input denormals flushed-to-0)
NaN support	Yes	Yes	Yes
Overflow and Infinity support	Yes	Yes	Yes
Flags	No	Yes	Yes
FMA	Yes	No	Yes
Square root	Software with low-latency FMA-based convergence	Hardware	Software only
Division	Software with low-latency FMA-based convergence	Hardware	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit + step
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit + step
log2(x) and 2^x estimates accuracy	23 bit	No	No

Single Precision Floating Point

	G80	SSE	IBM Altivec	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes, only clamps to max norm	Yes	Yes	No, infinity
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No

Products

Tesla S1070 1U System



4 Teraflops[1]

800 watts[2]

[1] single precision

[2] typical power

Tesla C1060 Board

957 Gigaflops[1]

160 Watts[2]

[1] single precision

[2] typical power

Building a 100TF datacenter

CPU 1U Server	Tesla 1U System
	
4 CPU cores	**4 GPUs: 960 cores**
0.07 Teraflop	4 Teraflops
$ 2000	$ 8000
400 W	800 W
1429 CPU servers	**25 CPU servers** **25 Tesla systems**
$ 3.1 M	$ 0.31 M
571 KW	27 KW

10x lower cost

21x lower power

Tesla Personal Supercomputer

Supercomputing Performance

- Massively parallel CUDA Architecture
- 960 cores. 4 TeraFlops
- 250x the performance of a desktop

Personal

- One researcher, one supercomputer
- Plugs into standard power strip

Accessible

- Program in C for Windows, Linux
- Available now worldwide under $10,000

C-for-CUDA SDK

Libraries:FFT, BLAS,CuDPP…
Example Source Code

Integrated CPU
and GPU C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing

CPU Host Code

CUDA
Driver

Debugger
Profiler

Standard C Compiler

GPU

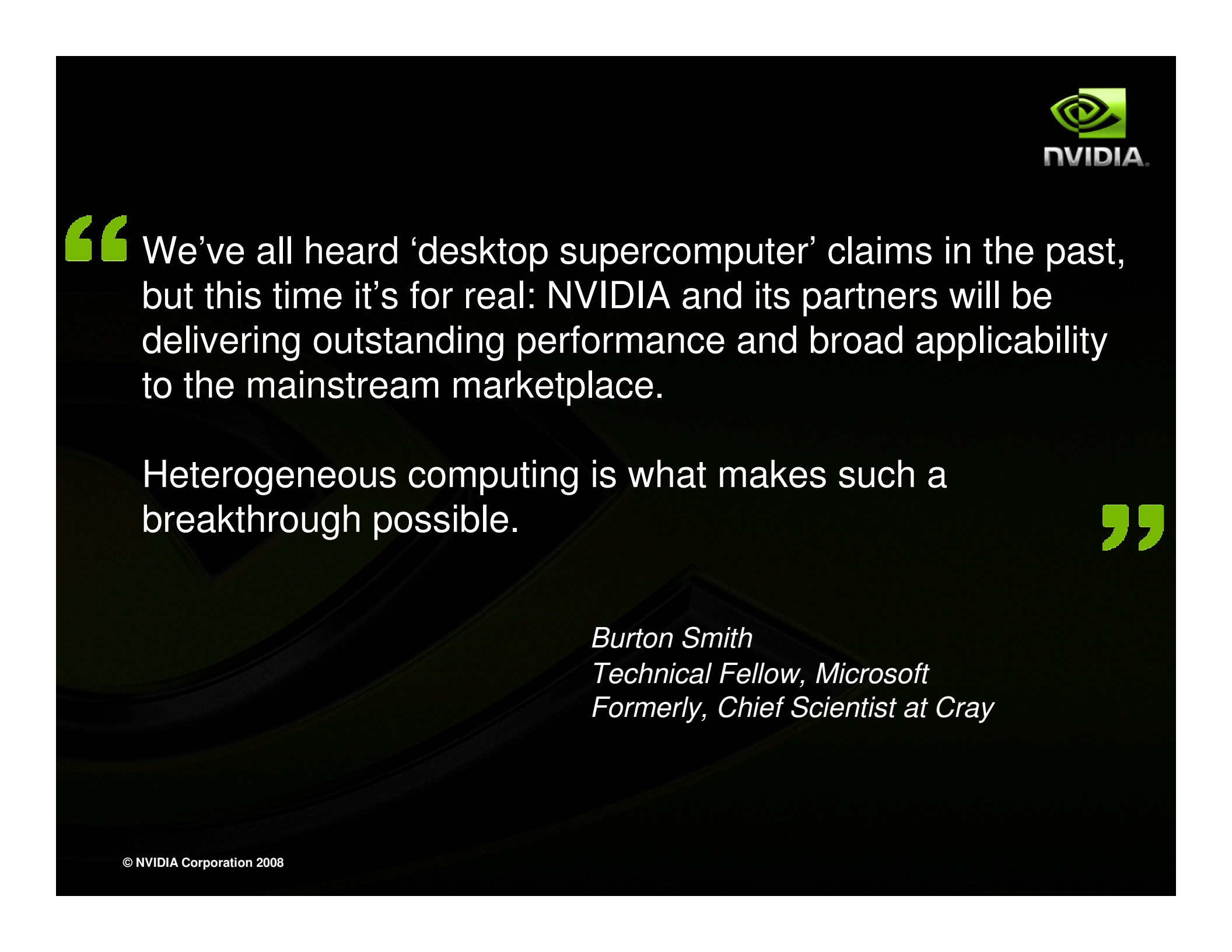
CPU

Quotes

GPUs have evolved to the point where many real world applications are easily implemented on them and run significantly faster than on multi-core systems.

Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs.

Jack Dongarra
Professor, University of Tennessee
Author of Linpack



“We’ve all heard ‘desktop supercomputer’ claims in the past, but this time it’s for real: NVIDIA and its partners will be delivering outstanding performance and broad applicability to the mainstream marketplace.

Heterogeneous computing is what makes such a breakthrough possible.”

Burton Smith
Technical Fellow, Microsoft
Formerly, Chief Scientist at Cray