ECSE-4965-01, Spring 2014

(modified from Stanford CS 193G)

Lecture 1: Introduction to Massively Parallel Computing

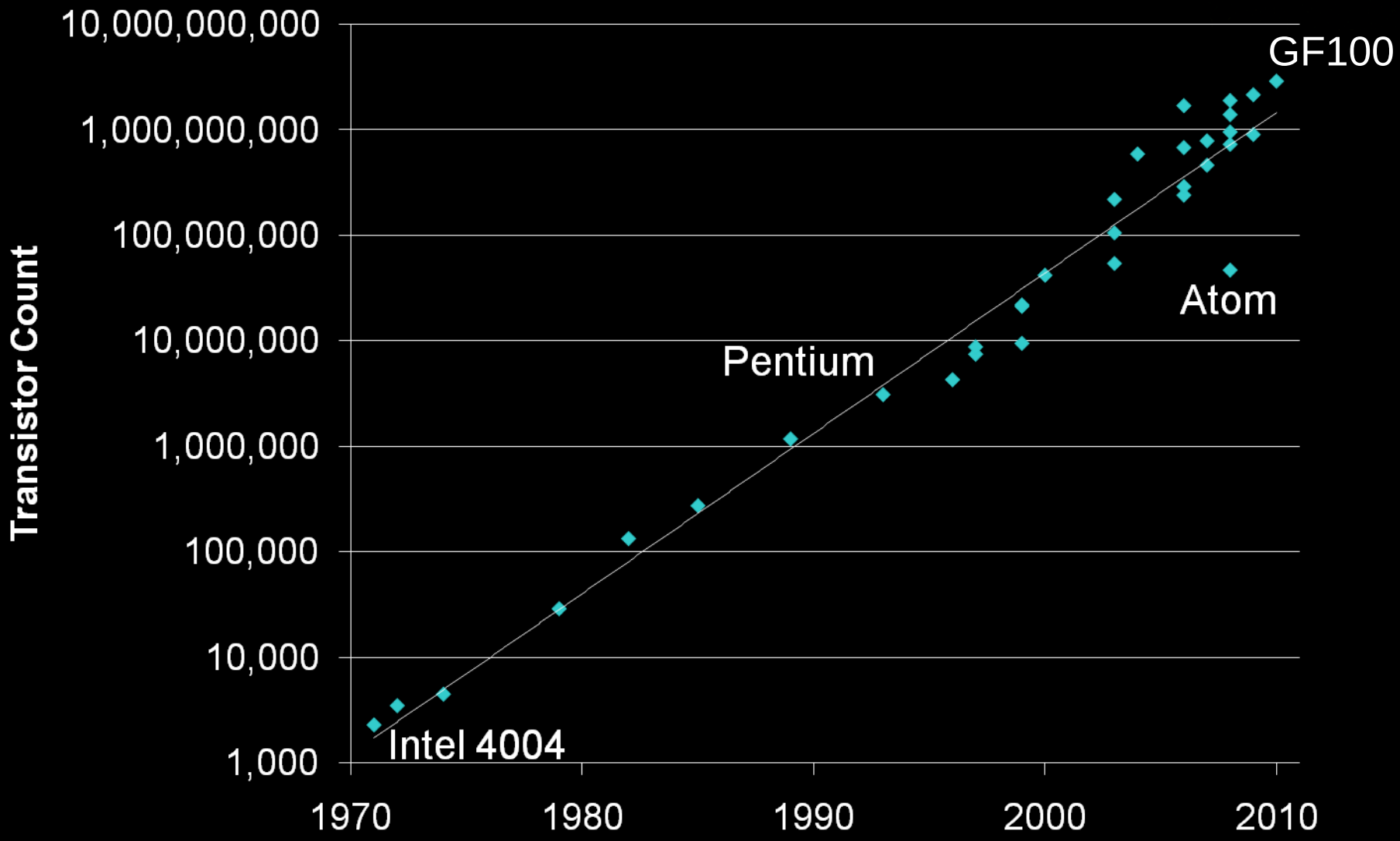
Moore's Law (paraphrased)

"The number of transistors on an integrated circuit doubles every two years."

– Gordon E. Moore

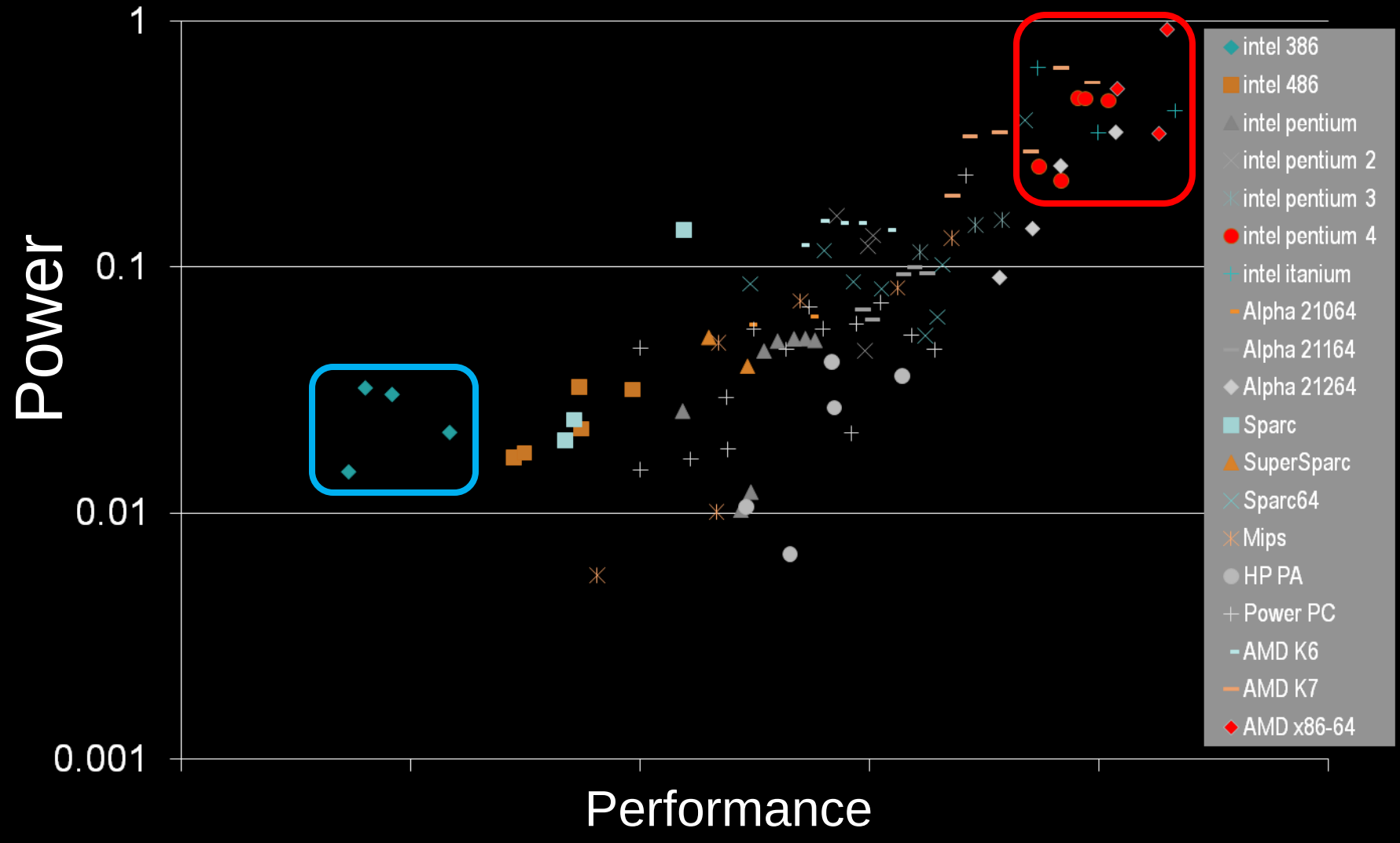
Note that one reason that this has remained valid for so long is that HW designers take it as a goal.

Moore's Law (Visualized)



Data credit: Wikipedia

Buying Performance with Power



(courtesy Mark Horowitz and Kevin Skadron)

Serial Performance Scaling is Over

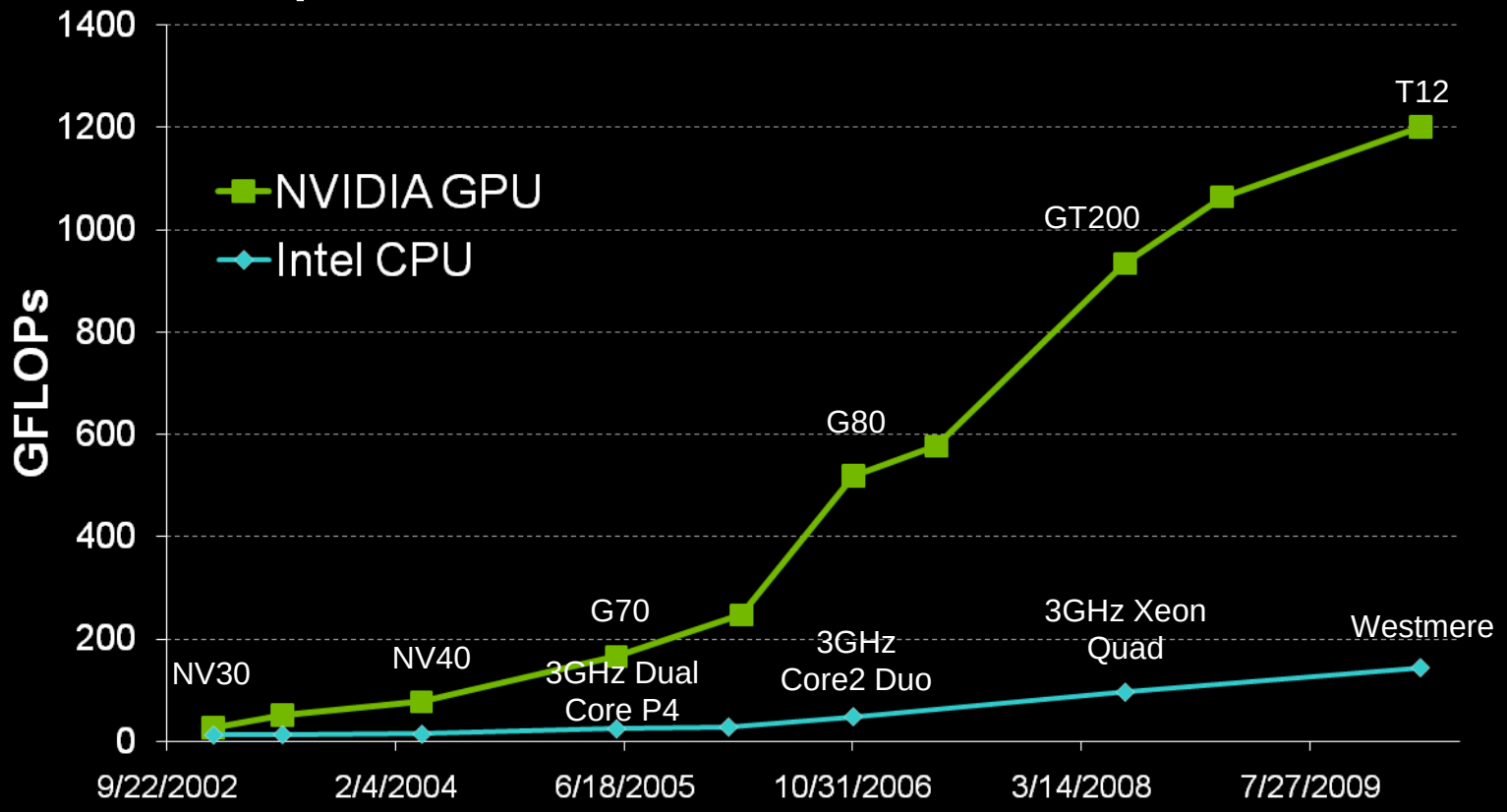
- Cannot continue to scale processor frequencies
  - no 10 GHz chips
- Cannot continue to increase power consumption
  - can’t melt chip
- Can continue to increase transistor density
  - as per Moore’s Law

How to Use Transistors?

- Instruction-level parallelism
  - out-of-order execution, speculation, …
  - vanishing opportunities in power-constrained world
- Data-level parallelism
  - vector units, SIMD execution, …
  - increasing … SSE, AVX, Cell SPE, Clearspeed, GPU
- Thread-level parallelism
  - increasing … multithreading, multicore, manycore
  - Intel Core2, AMD Phenom, Sun Niagara, STI Cell, NVIDIA Fermi, …

Why Massively Parallel Processing?

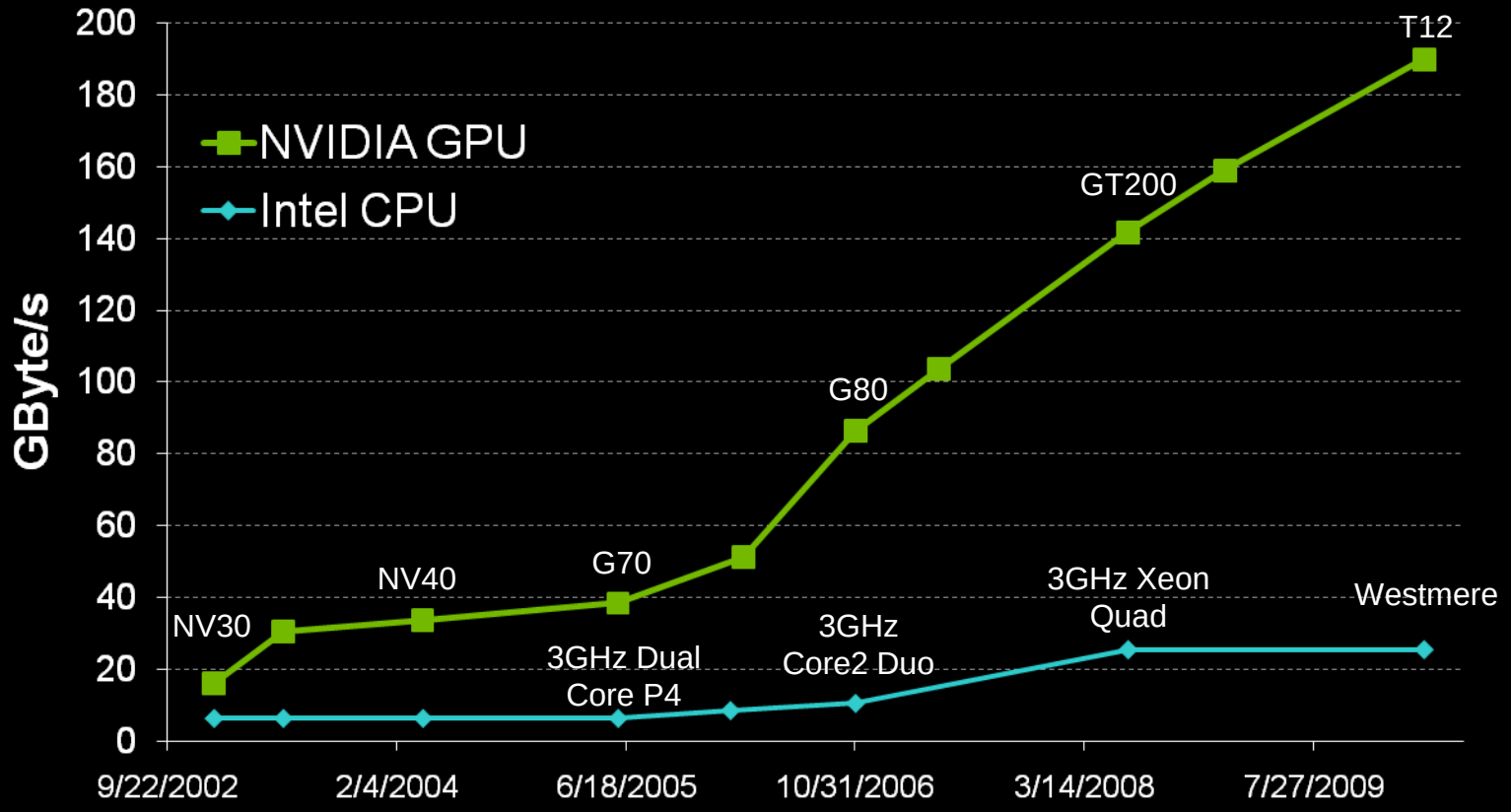
- A quiet revolution and potential build-up
  - Computation: TFLOPs vs. 100 GFLOPs



  - GPU in every PC – massive volume & potential impact

Why Massively Parallel Processing?

- A quiet revolution and potential build-up
  - Bandwidth: ~10x

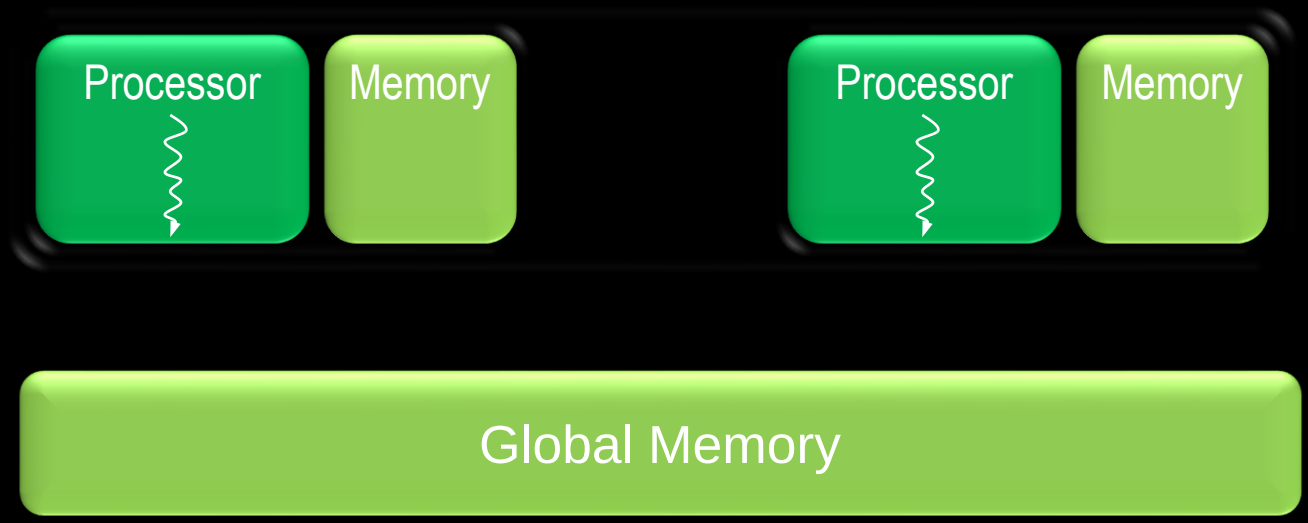


  - GPU in every PC – massive volume & potential impact

The “New” Moore’s Law

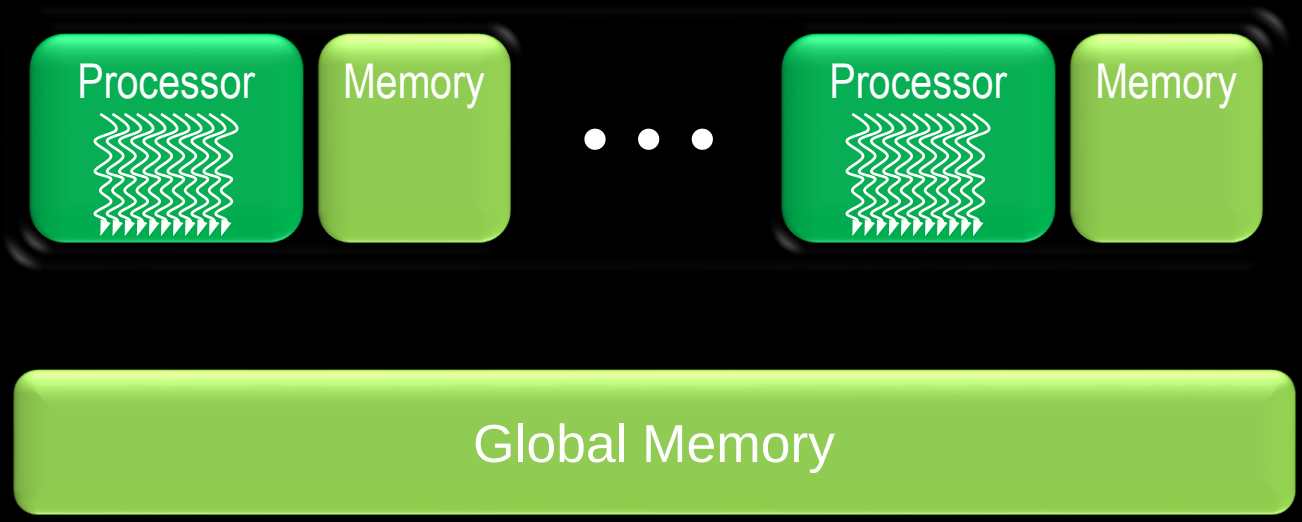
- Computers no longer get faster, just wider
- You *must* re-think your algorithms to be parallel !
- Data-parallel computing is most scalable solution
  - Otherwise: refactor code for ~~2 cores~~ ~~4 cores~~ ~~8 cores~~ 16 cores…
  - You will always have more data than cores – build the computation around the data

Generic Multicore Chip



- Handful of processors each supporting ~1 hardware thread
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)

Generic Manycore Chip



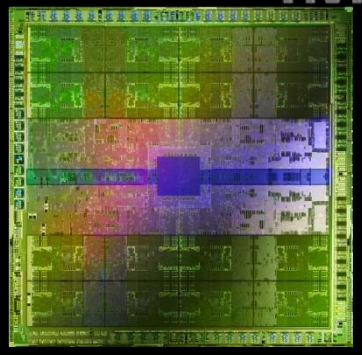
- Many processors each supporting many hardware threads
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)

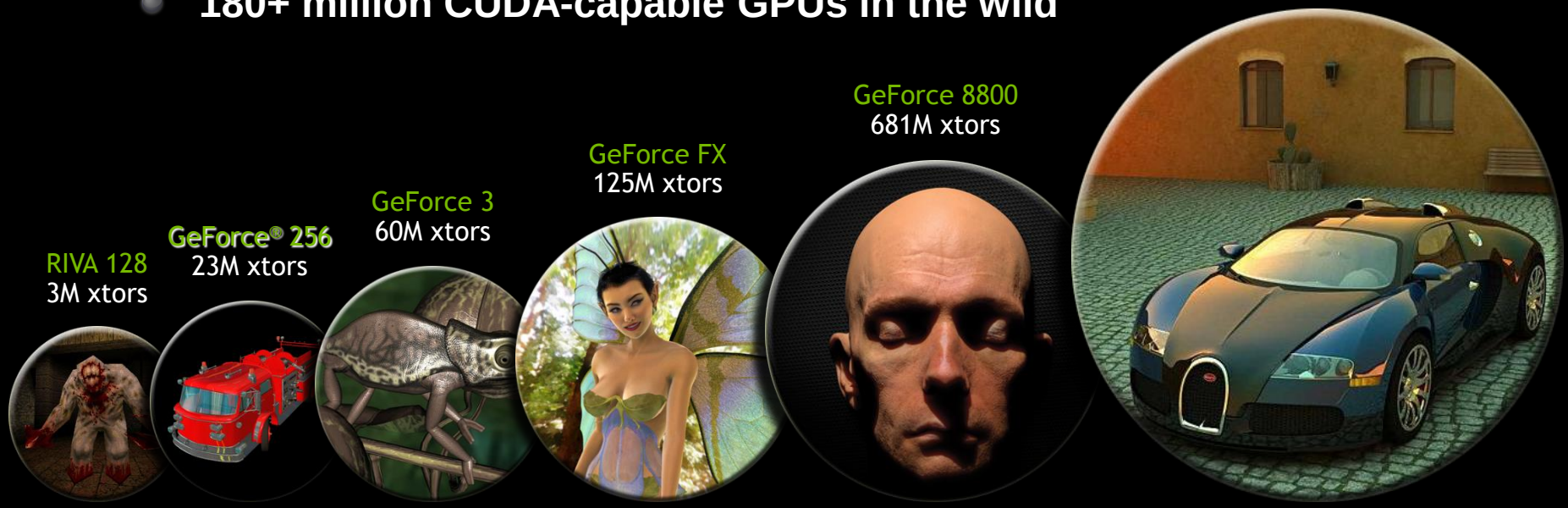
Enter the GPU

- Massive economies of scale
- Massively parallel

GPU Evolution

- **High throughput** computation
  - GeForce GTX 280: 933 GFLOP/s
- **High bandwidth** memory
  - GeForce GTX 280: 140 GB/s
- **High availability** to all
  - 180+ million CUDA-capable GPUs in the wild





Lessons from Graphics Pipeline

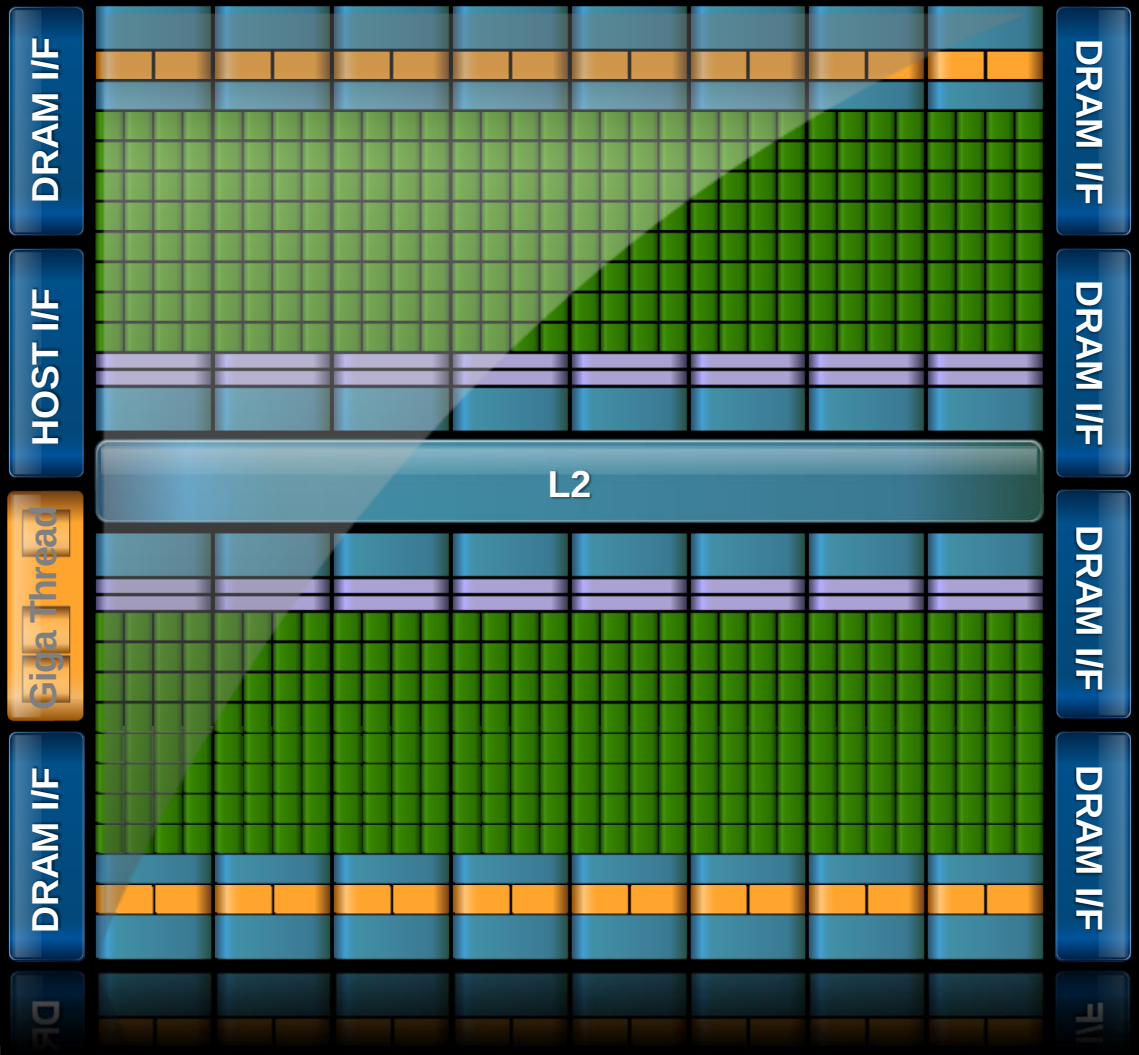
- **Throughput** is paramount
  - must paint every pixel within frame time
  - scalability
- Create, run, & retire **lots of threads** very rapidly
  - measured 14.8 Gthread/s on `increment()` kernel
- Use **multithreading** to hide latency
  - 1 stalled thread is OK if 100 are ready to run

Why is this different from a CPU?

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

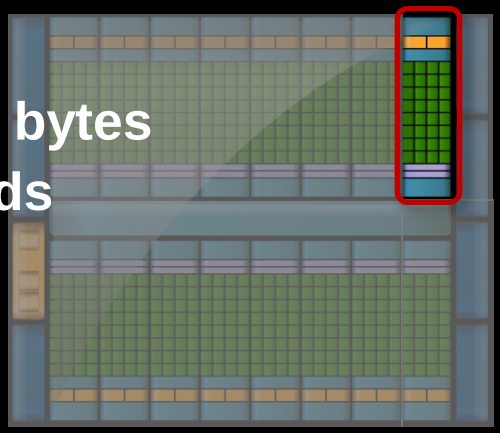
NVIDIA GPU Architecture

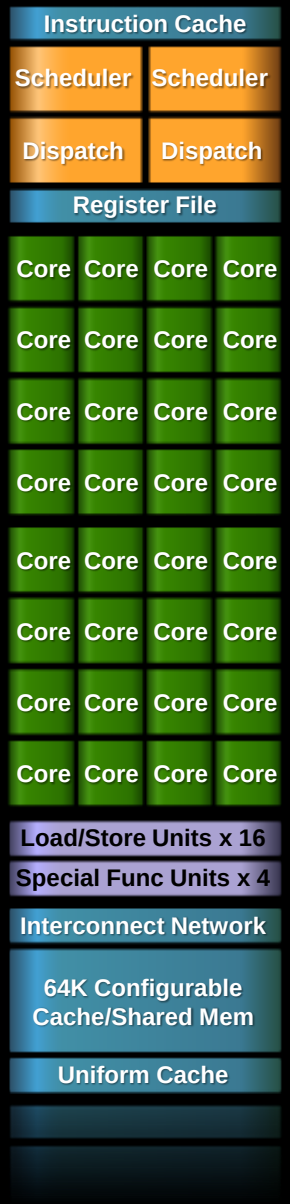
Fermi GF100



SM Multiprocessor

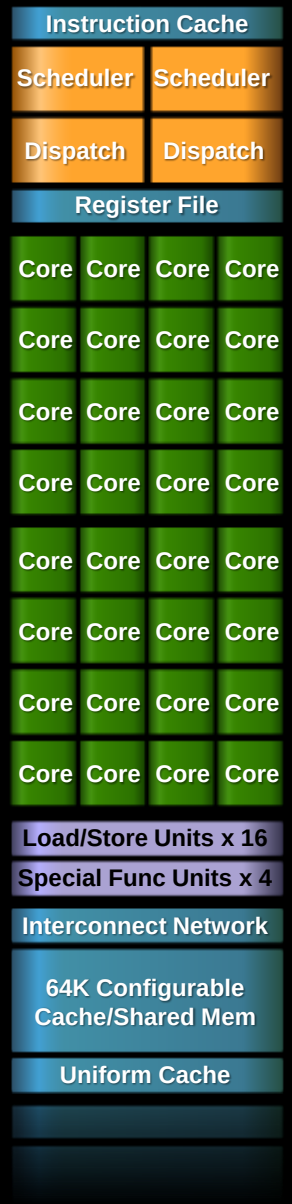
- 32 CUDA Cores per SM (512 total)
- 8x peak FP64 performance
  - 50% of peak FP32 performance
- Direct load/store to memory
  - Usual linear sequence of bytes
  - High bandwidth (Hundreds GB/sec)
- 64KB of fast, on-chip RAM
  - Software or hardware-managed
  - Shared amongst CUDA cores
  - Enables thread communication





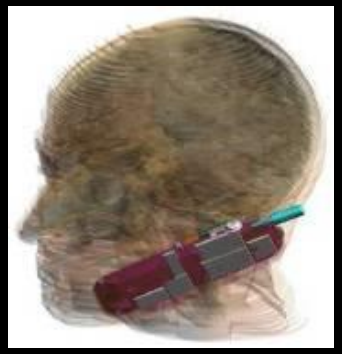
Key Architectural Ideas

- **SIMT (**Single Instruction Multiple Thread**) execution**
  - **threads run in groups of 32 called warps**
  - **threads in a warp share instruction unit (IU)**
  - **HW automatically handles divergence**
- **Hardware multithreading**
  - **HW resource allocation & thread scheduling**
  - **HW relies on threads to hide latency**
- **Threads have all resources needed to run**
  - **any warp not waiting for something can run**
  - **context switching is (basically) free**

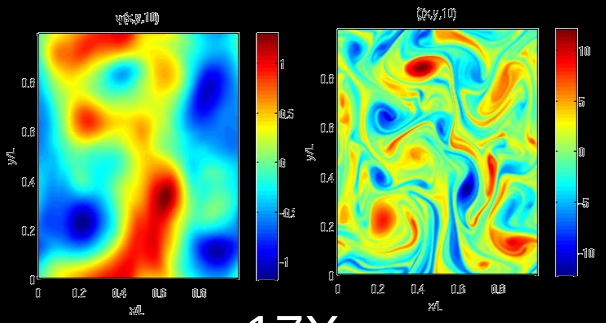


Enter CUDA

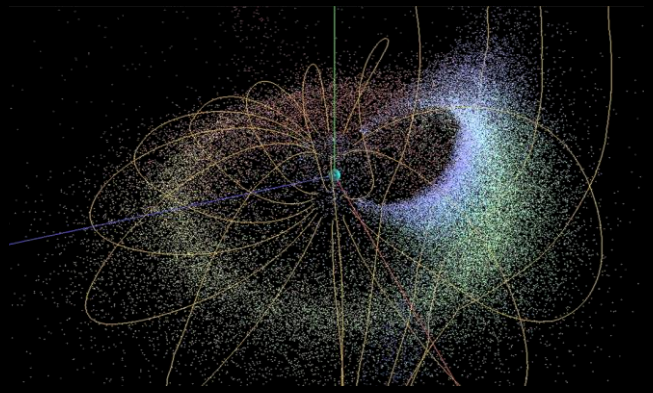
- Scalable parallel programming model
- Minimal extensions to familiar C/C++ environment
- Heterogeneous serial-parallel computing



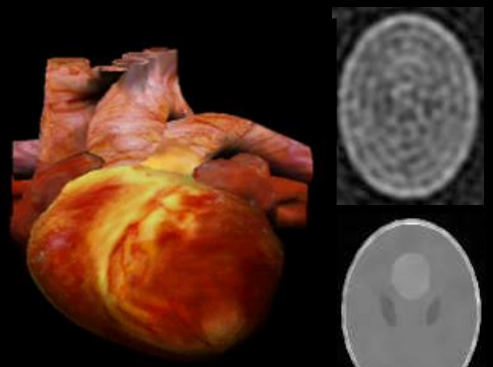
45X



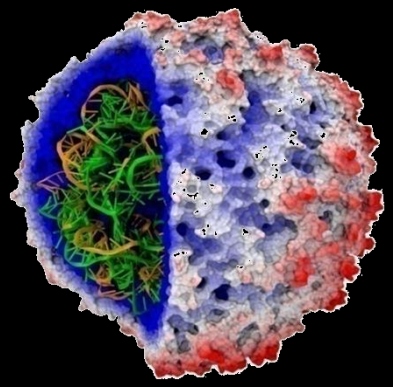
17X



100X

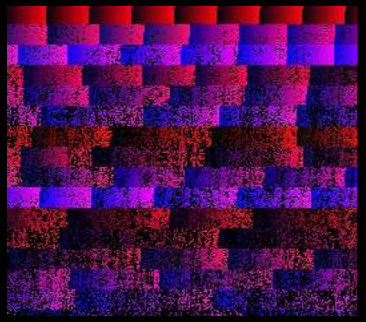


13–457x



110-240X

Motivation



35X

CUDA: Scalable parallel programming

- Augment C/C++ with minimalist abstractions
  - let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language
- Provide straightforward mapping onto hardware
  - good fit to GPU architecture
  - maps well to multi-core CPUs too
- Scale to 100s of cores & 10,000s of parallel threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization

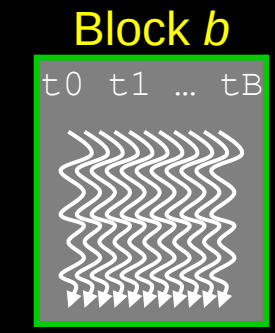
Key Parallel Abstractions in CUDA

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

Hierarchy of concurrent threads

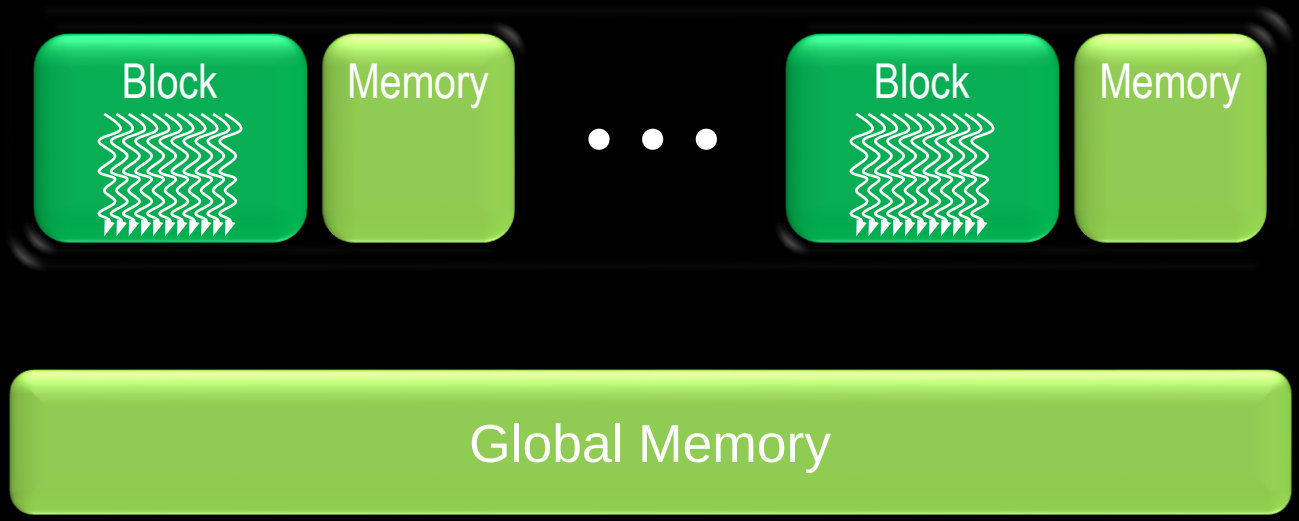
- Parallel kernels composed of many threads
  - all threads execute the same sequential program

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



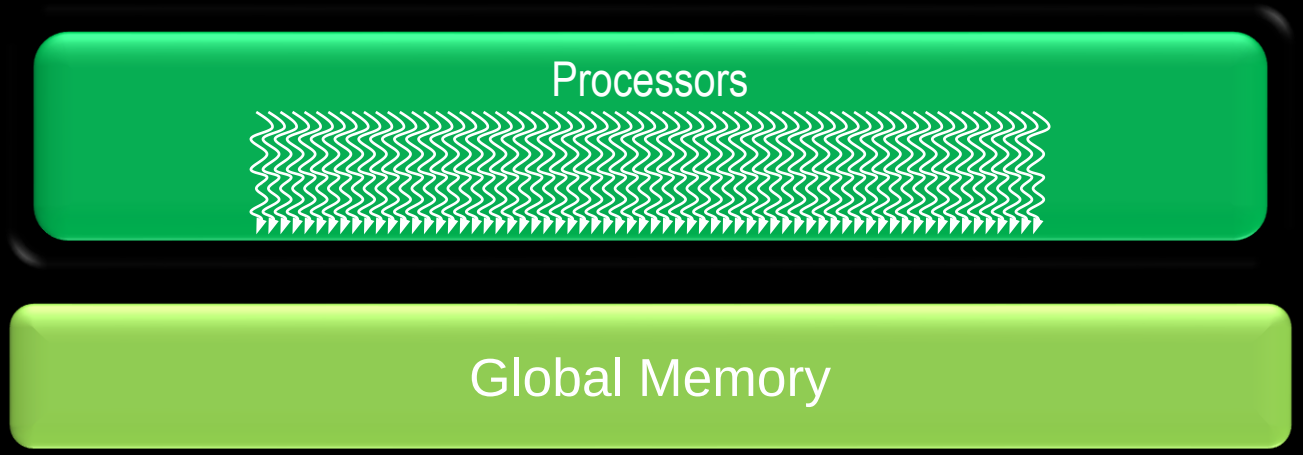
- Threads/blocks have unique IDs

CUDA Model of Parallelism



- CUDA virtualizes the physical hardware
  - thread is a virtualized scalar processor (registers, PC, state)
  - block is a virtualized multiprocessor (threads, shared mem.)
- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks should be independent

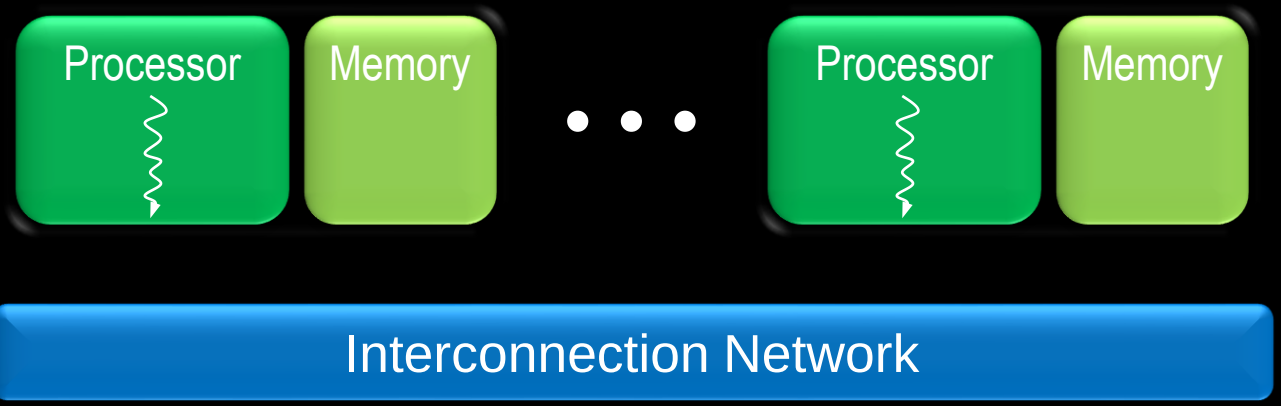
NOT: Flat Multiprocessor



- Global synchronization isn't cheap
- Global memory access times are expensive

- cf. PRAM (Parallel Random Access Machine) model

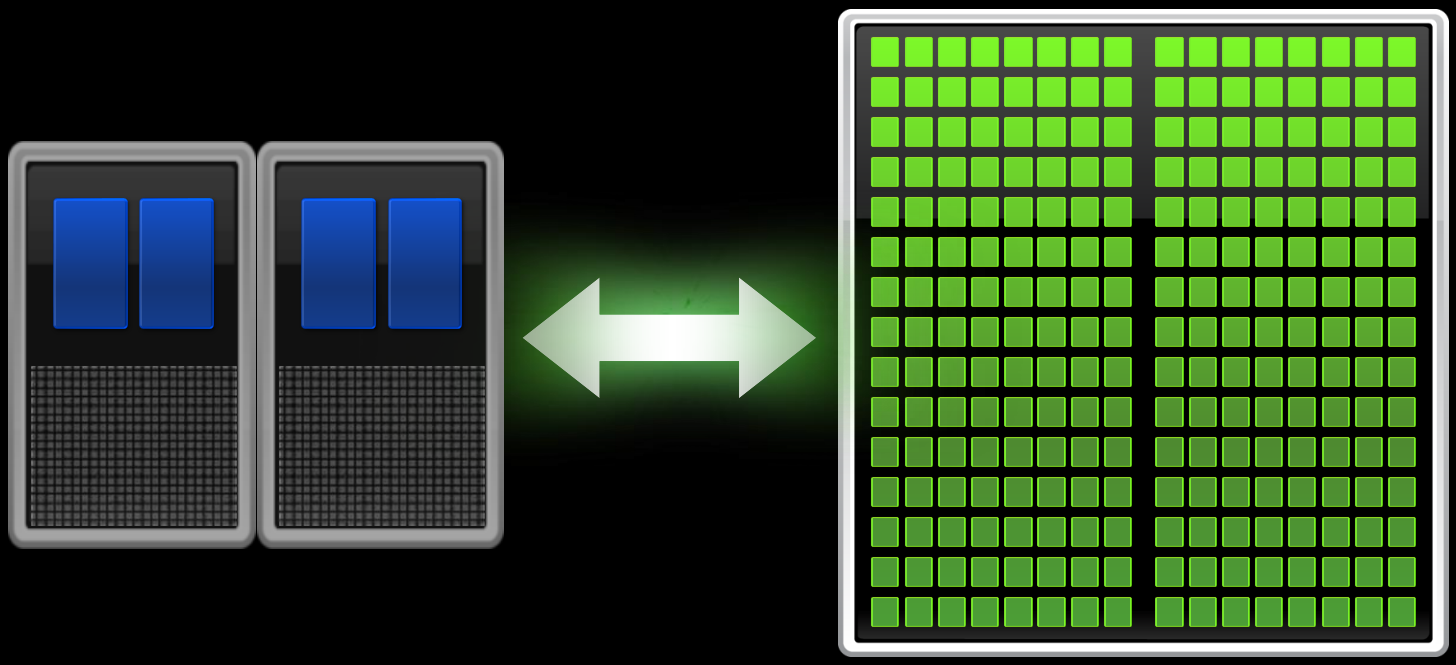
NOT: Distributed Processors



- Distributed computing is a different setting

- cf. BSP (Bulk Synchronous Parallel) model, MPI

Heterogeneous Computing



C for CUDA

- Philosophy: provide minimal set of extensions necessary to expose power
- Function qualifiers:

```
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

- Variable qualifiers:

```
__constant__ float my_constant_array[32];
__shared__   float my_shared_array[32];
```

- Execution configuration:

```
dim3 grid_dim(100, 50);  // 5000 thread blocks
dim3 block_dim(4, 8, 8); // 256 threads per block
my_kernel <<< grid_dim, block_dim >>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx; // Thread index
void __syncthreads(); // Thread synchronization
```

Example: vector_addition

Device Code

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // elided initialization code
    ...
    // Run N/256 blocks of 256 threads each
    vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: vector_addition


```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // elided initialization code
    ...
    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

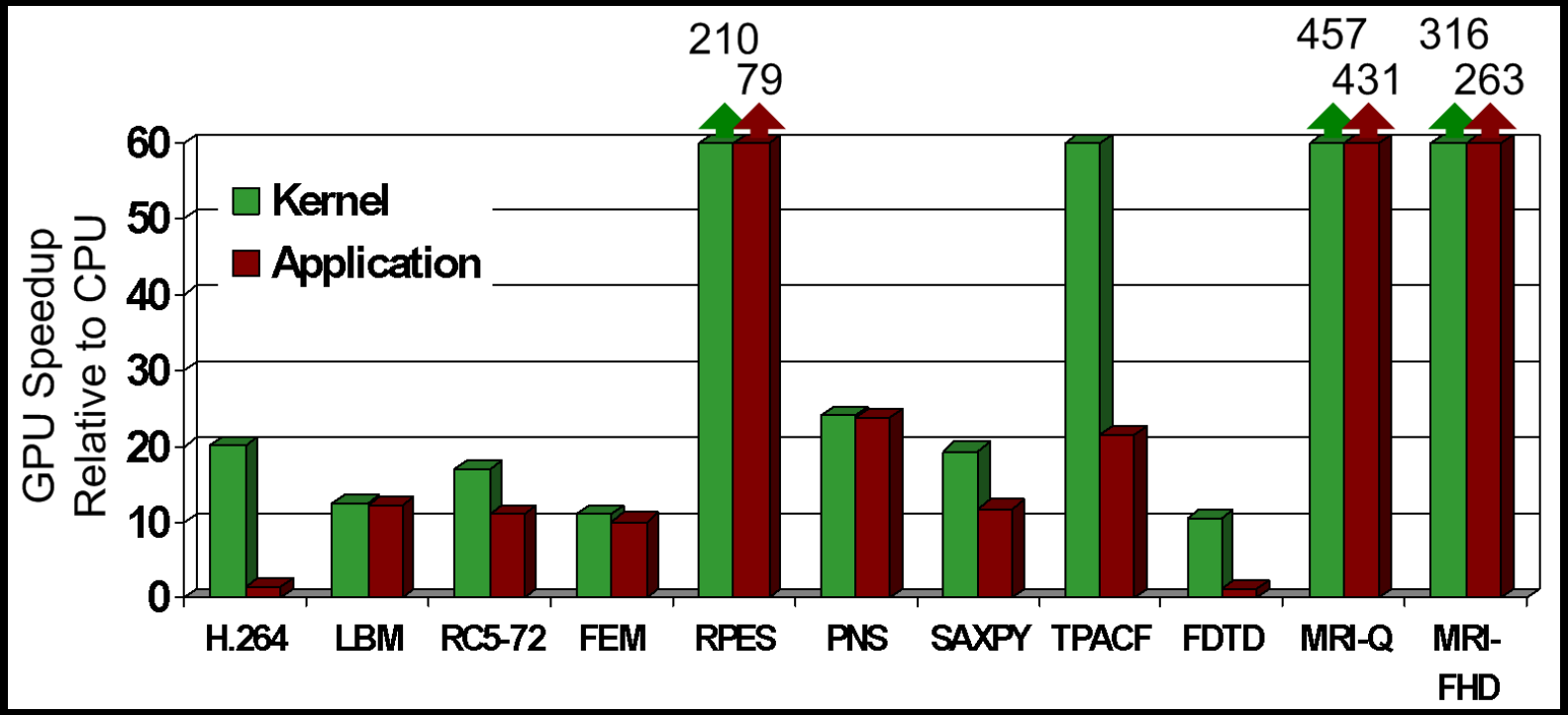
Example: Initialization code for vector_addition

```
// 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) );

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(d_A, d_B, d_C);
```

Previous Projects from UIUC ECE 498AL

Application	Description	Source	Kernel	% time
H.264	SPEC ‘06 version, change in guess vector	34,811	194	35%
LBM	SPEC ‘06 version, change to single precision and print fewer reports	1,481	285	>99%
RC5-72	Distributed.net RC5-72 challenge client code	1,979	218	>99%
FEM	Finite element modeling, simulation of 3D graded materials	1,874	146	99%
RPES	Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion	1,104	281	99%
PNS	Petri Net simulation of a distributed system	322	160	>99%
SAXPY	Single-precision implementation of saxpy, used in Linpack’s Gaussian elim. routine	952	31	>99%
TPACF	Two Point Angular Correlation Function	536	98	96%
FDTD	Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation	1,365	93	16%
MRI-Q	Computing a matrix Q, a scanner’s configuration in MRI reconstruction	490	33	>99%

Speedup of Applications



- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25× to 400× speedup if the function's data requirements and control flow suit the GPU and the application is optimized

Final Thoughts

- Parallel hardware is here to stay
- GPUs are massively parallel manycore processors
  - easily available and fully programmable
- Parallelism & scalability are crucial for success
- This presents many important research challenges
  - not to speak of the educational challenges

Machine Problem 0

- Work through tutorial codes
  - hello_world.cu
  - cuda_memory_model.cu
  - global_functions.cu
  - device_functions.cu
  - vector_addition.cu