Outline

- Visual Computing
- CUDA scalable parallel programming model
- Tesla parallel GPU architecture
- CUDA programming examples
- CUDA applications and performance
Visual Computing

- Visual computing is graphics + parallel computing that lets you visually interact with computed objects
- Video games – the first real-time visual computing applications
- Physics, video processing, image processing, 3D stereo
- Highly parallel computing on difficult problems

Graphics processing units (GPUs) evolved to provide rich real-time visual computing
GPUs today are highly parallel programmable processors

CUDA programming model enables parallel computing in C and inspired OpenCL and DX11 Compute Shaders
Parallelism is Scaling Rapidly

- CPUs and GPUs are parallel processors
  - CPUs now have 2, 4, 8, … processors
  - GPUs now have 32, 64, 128, 240, … processors

- Parallelism is increasing rapidly with Moore’s Law
  - Device count continues doubling every 18 – 24 months
  - Processor count continues to increase
  - Individual processor cores not getting much faster

- Challenge: Develop parallel application software
  - Scale software parallelism to use more and more processors
  - Same source for parallel GPUs and CPUs
GPU Sizes Require Scalable Software

32 SP Cores

128 SP Cores

240 SP Cores

© NVIDIA Corporation 2009
CUDA is a scalable parallel programming model

- Program runs on any number of processors without recompiling
- Compile the same program source to run on different platforms with widely different parallelism
- Map CUDA threads to GPU threads or CPU vector lanes

CUDA is standard C

- Write a program for one thread
- Instantiate it on many parallel threads
- Familiar programming model and language
CUDA Computing Sweet Spots

Parallel Applications:

- High arithmetic intensity:
  - Dense linear algebra, PDEs, $n$-body, finite difference, …

- High bandwidth:
  - Sequencing (virus scanning, genomics), sorting, database, …

- Visual computing:
  - Graphics, image processing, tomography, machine vision, …

- Computational modeling, science, engineering, finance, …
GPU Application Speedups

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 for portable video

17X
Matlab isotropic turbulence simulation using CUDA .mex file function

100X
Astrophysics nbody simulation

149X
Financial simulation of LIBOR Model with swaptions

47X
GLAME@lab: M-script API for Linear Algebra Operations on GPU

20X
Ultrasound medical imaging for cancer diagnostics

24X
Highly optimized object oriented molecular dynamics

30X
Cmatch exact string matching finds similar proteins & gene sequences
Pervasive CUDA Parallel Computing

- Over 100M CUDA-enabled GPUs
  - Makes parallel computing a commodity technology
- Over 100 Universities teaching CUDA
- Over 750 research papers
- Wide developer acceptance
  - Over 25K active CUDA developers
  - Download CUDA from www.nvidia.com/CUDA
  - A GPU “developer kit” costs ~$200 for 500 GFLOPS
- CUDA enables Tesla personal supercomputers
Some CUDA Design Goals

- Scale to 100s of cores, 1000s of parallel threads
  - Transparently with one source and same binary

- Let programmers focus on parallel algorithms
  - *Not* mechanics of a parallel programming language

- Enable heterogeneous systems (i.e. CPU+GPU)
  - CPU & GPU are separate devices with separate memories
Key Parallel Abstractions in CUDA

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

- Parallel **kernels** composed of many threads
  - All threads execute the same sequential program
  - Use parallel threads rather than sequential loops

- Threads are grouped into **thread blocks**
  - Threads in the same block can cooperate and share memory

- Blocks are grouped into **grids**
  - Threads and blocks have unique IDs
  - ThreadIdx
  - BlockIdx

```
Thread t
```

```
Block b
  t0 t1 ... tB
```

```
Grid
  Blocks 0
  Blocks 1
  Blocks 2
  ... Block m
```
What is a thread?

- Independent thread of execution
  - Has its own PC, variables in registers and local memory, thread state, thread ID, etc.
  - No implication about how threads are scheduled

- CUDA threads may be **physical** threads
  - as on NVIDIA GPUs

- CUDA threads may be **virtual** threads
  - as on multicore CPUs
  - Pick 1 thread block == 1 CPU core physical thread
What is a thread block?

- Thread block == **virtualized multiprocessor**
  - freely choose thread count 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
  - enables scalability to more and fewer parallel cores
Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD ... can easily deadlock

- Independence requirement gives **scalability**
CUDA 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];
}

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

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

Threads within block synchronize with **barriers**

... Step 1 ...

__syncthreads(); // synchronization barrier

... Step 2 ...

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

**Implicit global barrier between dependent kernels**

vec_minus<<<nblocks, blksz>>>(a, b, c);

// — implicit synchronization barrier —
vec_dot<<<nblocks, blksz>>>(c, c);
Levels of parallelism

- **Thread parallelism**
  - each thread is an independent thread of execution

- **Data parallelism**
  - across threads in a block
  - across blocks in a kernel

- **Task parallelism**
  - different blocks are independent
  - independent kernels

- Instruction-level and pipeline parallelism too
Memory model

Thread

Per-thread Local Memory

float LocalVar;

Kernel Sequence

Kernel 0

Kernel 1

Block

Per-block Shared Memory

__shared__ int SharedVar;

Per-app Device Global Memory

__device__ int GlobalVar;
CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live
  
  __global__ void KernelFunc(...);  // kernel callable from host
  __device__ void DeviceFunc(...);  // function callable on device
  __device__ int GlobalVar;        // variable in device memory
  __shared__ int SharedVar;       // in per-block shared memory

- Extend function invocation syntax for parallel kernel launch
  
  KernelFunc<<<500, 128>>>(...);  // 500 blocks, 128 threads each

- Special variables for thread identification in kernels
  
  dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;

- Intrinsics that expose specific operations in kernel code
  
  __syncthreads();  // barrier synchronization
CUDA: Features available on GPU

- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.

- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.

- Texture accesses in kernels
  ```
texture<float,2> my_texture; // declare texture reference

float4 texel = texfetch(my_texture, u, v);
```
Using per-block shared memory

- Variables shared across block
  
  ```c
  __shared__ int *begin, *end;
  ```

- Scratchpad memory

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

- Communicating values between threads

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

Host memory

cudaMemcpy()

Device 0 memory

Device 1 memory
CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  
  ```
  cudaMemcpy(), cudaMemcpy2D(), ...
  ```

- Explicit memory copy for host ↔ device, device ↔ device
  
  ```
  cudaMemcpy(), cudaMemcpy2D(), ...
  ```

- Texture management
  
  ```
  cudaBindTexture(), cudaBindTextureToArray(), ...
  ```

- OpenGL & DirectX interoperability
  
  ```
  cudaGLMapBufferObject(), cudaD3D9MapVertexBuffer(), ...
  ```
CUDA: 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);
CUDA Uses Extensive Multithreading

CUDA **threads** express fine-grained data parallelism
- Map threads to GPU threads or CPU vector elements
- Virtualize the processors
- Rethink your algorithms to be aggressively parallel

CUDA **thread blocks** express coarse-grained parallelism
- Map blocks to GPU thread arrays or CPU threads
- Scale transparently to any number of processors

**GPUs** execute thousands of lightweight threads
- One graphics thread computes one pixel fragment
- One CUDA thread computes one result (or several results)
- Provide hardware multithreading & zero-overhead scheduling
CUDA Computing with Tesla T10

- 240 SP processors at 1.4 GHz: 1 TFLOPS peak
- 128 threads per processor: 30,720 threads total
Tesla T10 GPU Parallel Processor

- Manycore parallel processor
  - 240 thread processors at 1.4 GHz
  - 1.4 billion transistors
  - 1 TFLOPS IEEE single precision

- Massively multithreaded
  - 128 threads per processor
  - 30,720 threads total
  - Hardware thread management
  - Zero-overhead scheduling

- Tesla unified scalable architecture
  - Parallel computing
  - 3D Graphics rendering
  - T10 scales 2x beyond Tesla 8-series
Implementation

- TSMC 55nm
- 1.4 Billion Transistors
- 2236 Ball BGA

- 1 TeraFLOPS SP / 84 GigaFLOPS DP
  - 1.4 GHz Processor Clock
  - 1.1 GHz Memory Clock
- Up to 4GB on-board Memory
Tesla 10-Series Architecture

- Tesla architecture unifies graphics and computing
- 240 multithreaded thread processors at 1.4 GHz
- 4 GB DRAM in 8 x 64-bit partitions, 102 GB/s
- Single precision 32-bit IEEE at 1 TFLOPS peak
- Double precision 64-bit IEEE at 86 GFLOPS peak
- Up to 30K concurrent thread contexts, 128 per thread processor
- Threads virtualize scalable processors and memories
T10 SM Multithreaded Multiprocessor

- 8 SP Thread Processors
  - IEEE 754 32-bit floating point
  - 32-bit and 64-bit integer
  - 2K 32-bit registers per SP
- 2 SFU Special Function Units
- 1 DP Double Precision Unit
  - IEEE 754R 64-bit floating point
  - Fused multiply-add
- Scalar register-based ISA
- Multithreaded Instruction Unit
  - 1024 threads, hardware managed
  - Independent thread execution
  - Zero-overhead thread scheduling
- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store
IEEE 754-2008 64-bit results for all DP instructions
- DADD, DMUL, DFMA, DMAX, DMIN, DSET, DtoF, FtoD, Dtol, ItoD
- Rounding, denorms, NaNs, +/- Infinity

IEEE 754-2008 Fused multiply-add (DFMA)
- \( D = A \times B + C \); with no loss of precision in the add
- DDIV and DSQRT software use FMA-based convergence

IEEE 754-2008 rounding: nearest even, zero, +inf, -inf

Full-speed denormalized operands and results

No exception flags

Peak DP (DFMA) performance 86 GFLOPS at 1.44 GHz
SIMT Multithreaded Execution

- **SIMT**: Single/Instruction Multi-Thread applies instruction to independent threads
- **Warp**: the set of 32 parallel threads that execute a SIMT instruction
- SIMT provides flexible single-thread programming with SIMD efficiency
- Hardware implements zero-overhead warp and thread scheduling
- Each SP thread processor executes up to 128 concurrent threads
- SIMT threads execute independently
- Fast barrier synchronization
Coalesced Memory Access

- Individual threads access independent addresses
- A thread loads/stores 1, 2, 4, 8, 16 B
- For parallel threads in a warp, load/store hardware coalesces individual thread accesses into one or more memory block accesses
- Implements vector scatter/gather
- Coalescing scales gracefully with the number of unique memory blocks accessed
Execution Model – CUDA

- **Local Memory:** per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory:** per-block
  - Shared by threads of CTA
  - Inter-thread communication using barrier synchronization
- **Global Memory:** per-application
  - Shared by all threads
  - Inter-Grid communication

Sequential Grids in Time

© NVIDIA Corporation 2009
Execution Model – Graphics Shaders

- Local Registers: per Pixel
  - Private per pixel
- Local Memory: per Pixel
  - Indexed arrays, register spill
- Planes/Textures: per Warp
  - Define surface-space inputs
  - Array of 1D/2D/3D data arrays
- Target Images: per Grid
  - Array of 2D surfaces

Pixel Thread
- Local Memory
- Registers

Pixel Warp
- Plane Equations
- Texture

Pixel Grid 0

Pixel Grid 1

Target Images
Compiling CUDA for GPUs

Generic

C/C++ CUDA Application

NVCC

PTX Code

CPU Code

Specialized

PTX to Target Translator

Target device code

© NVIDIA Corporation 2009

GPU Computing  Mar. 5, 2009
SAXPY C code: serial and CUDA

```c
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);
```

```c
__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);
```
SAXPY in PTX 1.0 ISA

// Calculate i from thread/block IDs
ctaid.x = blockid;
ntid.x = blocksize;
tid = tid;
i = blockid * blocksize + tid;

// Nothing to do if n ≤ i
n = N;
if n > i then
    offset = i * 4;
y = y + offset;
x = x + offset;

// Compute and store alpha*x[i] + y[i]
alpha = ALPHA;
y_i = y + offset;
x_i = x + offset;

alpha * x_i + y_i = y_i;
y_i = y + offset;

exit;
Example: Parallel Reduction

Summing up a sequence with 1 thread:

```c
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
Parallel Reduction

Values (shared memory)

10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 1
Stride 1

Thread IDs
0 1 2 3 4 5 6 7

Values
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2

Step 2
Stride 2

Thread IDs
0 1 2 3

Values
18 1 7 -1 -2 -2 8 5 4 -3 9 7 11 11 2 2

Step 3
Stride 4

Thread IDs
0 1

Values
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

Step 4
Stride 8

Thread IDs
0

Values
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

© NVIDIA Corporation 2009
Parallel Reduction

- Tree-based approach used within each thread block

- Need to be able to use multiple thread blocks
  - To process very large arrays
  - Use all GPU multiprocessors effectively
  - Each thread block reduces a portion of the array

- But how to communicate partial results between thread blocks?
The global synchronization “myth”

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively

Problem: GPU has limited resources
- GPU has M multiprocessors
- Each multiprocessor can support a limited # of blocks, b
- If total # blocks $B > M \times b$ …, global sync can deadlock

Also, GPUs rely on large amount of *independent* parallelism to cover memory latency
- Global synchronization destroys independence, and thus impairs scalability
Solution: kernel decomposition

- Avoid global synchronization by decomposing the computation into multiple kernel invocations

- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

Level 0: 8 blocks
Level 1: 1 block
Parallel Reduction

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```
NVIDIA PhysX

PhysX – full physics simulation library and SDK
  - Rigid body
  - Fluid flow
  - Cloth
  - Softbodies

APEX – Adaptive Physics Extensions
  - Middleware to make scalable games
  - Destruction authoring
<table>
<thead>
<tr>
<th>Resources, examples, and pointers for CUDA developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Zone: <a href="http://www.nvidia.com/cuda">www.nvidia.com/cuda</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Latest CUDA News</th>
<th>Parallel Computing &amp; Nvision 2008 – Save $100, Sign Up by June 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Algorithms-by-Block Made easy</td>
<td>Low Viscosity Flow Simulations for Animation</td>
</tr>
<tr>
<td>PyCUDA</td>
<td>Towards Acceleration of Fault Simulation</td>
</tr>
<tr>
<td>Efficient Computation of Sum Products on GPUs</td>
<td>Optical Flow Algorithm using CUDA and OpenCV</td>
</tr>
<tr>
<td>3 min (3.0x) Base</td>
<td>9 min (0.5x) Run</td>
</tr>
<tr>
<td>Silicon Informatics Protein Docking</td>
<td>Biomedical Image Analysis</td>
</tr>
<tr>
<td>23 x</td>
<td>13 x</td>
</tr>
<tr>
<td>SciFinance® Speeds Financial Results with Parallel Computing</td>
<td>Relational Joins on Graphics Processors</td>
</tr>
<tr>
<td>30 x</td>
<td>7 x</td>
</tr>
<tr>
<td>Jaccuda</td>
<td>Tomographic Reconstruction</td>
</tr>
<tr>
<td>43 x</td>
<td></td>
</tr>
</tbody>
</table>