Who is NVIDIA?
THE WORLD LEADER IN VISUAL COMPUTING
Vision: Mainstream Parallel Programming

- Enable more programmers to write parallel software
- Give programmers the choice of language to use
- Embrace and evolve standards in key languages
What is OpenPower?
OpenPOWER, a catalyst for Open Innovation

Market Shifts

• Moore’s law no longer satisfies performance gain
• Growing workload demands
• Numerous IT consumption models
• Mature Open software ecosystem

New Open Innovation

Open Development
open software, open hardware

Collaboration of thought leaders
simultaneous innovation, multiple disciplines

Performance of POWER architecture
amplified capability

• Rich software ecosystem
• Spectrum of power servers
• Multiple hardware options
• Derivative POWER chips

Feeds back … resulting in client choice

OpenPOWER is an open development community, using the POWER Architecture to serve the evolving needs of customers.
What is GPU Computing?
Accelerated Computing
GPU Accelerates Tasks Running on CPUs

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Many Parallel Tasks

10x Performance
5x Energy Efficiency
Low Latency or High Throughput?

CPU
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
How GPU Acceleration Works

Application Code

Compute-Intensive Functions

Small slice of Code

Rest of Sequential CPU Code

GPU

CPU
Basic GPU Layout (Fermi & Kepler)
Heterogeneous Computing

- Terminology:
  - Host: The CPU and its memory (host memory)
  - Device: The GPU and its memory (device memory)
Execution Model
Anatomy of a CUDA Application

- **Serial** code executes in a Host (CPU) thread
- **Parallel** code executes in many Device (GPU) threads across multiple processing elements
Some Terminology

Kernel - A function which runs on the GPU
- A kernel is launched on a grid of thread blocks.
- The grid and block size are called the launch configuration.

Global Memory - GPU’s on-board DRAM

Shared Memory - On-chip fast memory local to a thread block
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
CUDA Execution Model

- **Thread**: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel

- **Thread Block**: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange

- **Grid**: a collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive
Kernel Execution

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

CUDA thread

CUDA thread block

CUDA kernel grid

CUDA core

CUDA Streaming Multiprocessor

CUDA-enabled GPU

- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
IDs and Dimensions

- A kernel is launched as a grid of blocks of threads

**Built-in variables:**
- threadIdx
- blockIdx
- blockDim
- gridDim
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory that they all use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs
Warps

Blocks are divided into 32 thread wide units called warps
- Size of warps is implementation specific and can change in the future

The SM creates, manages, schedules and executes threads at warp granularity
- Each warp consists of 32 threads of contiguous threadIds

All threads in a warp execute the same instruction
- If threads of a warp diverge the warp serially executes each branch path taken

When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible
Memory Model
# Heterogeneous Computing

### Parallel Code

```c
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__
void stencil_1d(int* in, int* out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int* x, int n) {
    fill_n(x, n, 1);
}

int main(void) {

    int* in, *out;

    // host copies of a, b, c
    int* d_in, *d_out;

    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int*)malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void**)&d_in, size);
    cudaMalloc((void**)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
```

### Serial Code

```c
void fill_ints(int* x, int n) {
    fill_n(x, n, 1);
}

int main(void) {

    int* in, *out;

    // host copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int*)malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    return 0;
}
```
Memory Management

- Host and device memory are separate entities
  - **Device** pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - **Host** pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Multiple Memory Spaces

- Memory is allocated with locality
  - `cudaMalloc(&d_ptr)` - device
    - `d_ptr` cannot be dereferenced from the CPU
  - `Ptr = Malloc(); ptr=new, cudaMallocHost(&ptr)` - CPU
    - `Ptr` cannot be dereferenced from the GPU

- Zero Copy and GPUDirect allow you to circumvent (functionally)
  - Future architectures will improve.

- As of CUDA 4.0, virtual address range of CPU and GPU are unique
  - Can determine where the target of a pointer lives
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory

- **All blocks:**
  - Global memory
Putting it together
Vector Add: GPU’s Hello World

- GPU is *parallel computation* oriented.
  - Vector add is a very simple parallel algorithm.

- Problem: \( C = A + B \)
  - \( C, A, B \) are length \( N \) vectors

```c
void vecAdd(int n, float * a, float * b, float * c)
{
    for(int i=0; i<n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```
GPU is *parallel computation* oriented.

Vector add is a very simple parallel algorithm.

**Problem:** \( C = A + B \)

- \( C, A, B \) are length \( N \) vectors

```c
void main()
{
    int N = 1024;
    float * a, *b, *c;
    a = (float*)malloc(N*sizeof(float));
    b = (float*)malloc(N*sizeof(float));
    c = (float*)malloc(N*sizeof(float));
    memset(c, 0, N*sizeof(float));
    init_rand_f(a, N);
    init_rand_f(b, N);
    vecAdd(N, a, b, c);
}

void vecAdd(int n, float * a, float * b, float * c)
{
    for(int i=0; i<n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```
Moving Computation to the GPU

Step 1: **Identify parallelism.**
- Design problem decomposition

Step 2: Write your GPU Kernel

Step 3: Setup the Problem

Step 4: Launch the Kernel

Step 5: Copy results back from GPU

Remember: big font means important
Vector Add

Step 1: Parallelize
- Identify parallelism.
  - \( c[i] \) depends only on \( a[i] \) and \( b[i] \).
  - \( c[i] \) is not used by any other calculation
  - \( c[i] \) can be computed in parallel

Assign Units of Work
- Each thread will compute one element of \( c \).
- Will use a 1D grid of 1D threadblocks.

```c
void vecAdd(int n, float * a, float * b, float * c)
{
    for(int i=0; i<n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```
IDs and Dimensions

**Built-in variables:**
- `threadIdx.[x y z]`
  - thread index within a thread block
- `blockIdx.[x y z]`
  - block index within the grid.
- `blockDim.[x y z]`
  - Number of threads in each block.
- `gridDim.[x y z]`
  - Number of blocks in the grid.
Parallelization of VecAdd

| Thread | 0 | 1 | 2 | ... | 255 | 256 | ... | 511 | 512 | ... | 767 | ... | N |
|--------|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|
Parallelization of VecAdd

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>255</th>
<th>256</th>
<th>...</th>
<th>511</th>
<th>512</th>
<th>...</th>
<th>N</th>
</tr>
</thead>
</table>

ThreadBlock 0

ThreadBlock 1

ThreadBlock 2

...
Parallelization of VecAdd

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>255</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thread</th>
<th>256</th>
<th>…</th>
<th>511</th>
<th>…</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs</td>
<td>a[256]</td>
<td>a[511]</td>
<td>a[512]</td>
<td>a[N]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b[256]</td>
<td>b[511]</td>
<td>b[512]</td>
<td>b[N]</td>
<td></td>
</tr>
<tr>
<td>outputs</td>
<td>c[256]</td>
<td>c[511]</td>
<td>c[512]</td>
<td>c[N]</td>
<td></td>
</tr>
</tbody>
</table>

work index $i = threadIdx.x + blockIdx.x \times blockDim.x$;

- index of the thread within a thread block
- index of the threadblock within the grid
- number of threads within each block
Vector Add: GPU’s Hello World

Step 2: Make it a GPU Kernel

Identify this function as something to be run on the GPU.

```c
__global__ void vecAdd(int n, float * a, float * b, float * c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n) {
        c[i] = a[i] + b[i];
    }
}
```

i is a different value for each thread.

Protect against invalid access if too many threads are launched.
Step 3: Setup the problem

```c
void main()
{
    int N = 1024;
    float *a, *b, *c;
    float *devA, *devB, *devC;
    a = (float*)malloc(N*sizeof(float));
    b = (float*)malloc(N*sizeof(float));
    c = (float*)malloc(N*sizeof(float));
    cudaMalloc(&devA, N*sizeof(float));
    cudaMalloc(&devB, N*sizeof(float));
    cudaMalloc(&devC, N*sizeof(float));
    memset(c, 0, N*sizeof(float));
    init_rand_f(a, N);
    init_rand_f(b, N);
    cudaMemcpy(devA, a, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(devB, b, N*sizeof(float), cudaMemcpyHostToDevice);
}
```
Step 4: Launch the GPU Kernel

```c
void main()
{
    ...
    vecAdd<<<(N+127)/128, 128>>>(N, devA, devB, devC);
    ...
}
```

Angle Brackets: Specify **launch configuration** for the kernel.

- **call function by name as usual**
- **Normal parameter passing syntax. Note that devA, devB, and devC are device pointers.** They point to memory allocated on the GPU.
- **First argument is the number of thread blocks (rounding up)**
- **Second argument is the shape of (i.e., number of threads in) each thread block**

**Angle Brackets:** Specify the launch configuration for the kernel.

- **First argument** is the number of thread blocks (rounding up).
- **Second argument** is the shape of each thread block (i.e., number of threads in).

```
vecAdd<<<(N+127)/128, 128>>>(N, devA, devB, devC);
```

Normal parameter passing syntax. Note that devA, devB, and devC are device pointers. They point to memory allocated on the GPU.
Step 5: Copy data back.

```c
void main()
{
  ...
  cudaMemcpy(c, devC, N*sizeof(float), cudaMemcpyDeviceToHost);
  ...
}
```
A slightly more advanced version
Consider applying a 1D stencil to a 1D array of elements
- Each output element is the sum of input elements within a radius

If radius is 3, then each output element is the sum of 7 input elements:
Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory

- Extremely fast on-chip memory, user-managed

- Declare using `__shared__`, allocated per block

- Data is not visible to threads in other blocks
Implementing With Shared Memory

- Cache data in shared memory
  - Read \((\text{blockDim.x} + 2 \times \text{radius})\) input elements from global memory to shared memory
  - Compute \(\text{blockDim.x}\) output elements
  - Write \(\text{blockDim.x}\) output elements to global memory

Each block needs a **halo** of \(\text{radius}\) elements at each boundary.
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
            in[gindex + BLOCK_SIZE];
    }
Stencil Kernel

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
Data Race!

- The stencil example will not work...

- Suppose thread 15 reads the halo before thread 0 has fetched it...

```c
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

- **Store at temp[18]**
- **Skipped, threadIdx > RADIUS**
- **Load from temp[19]**
void __syncthreads();

Synchronizes all threads within a block
- Used to prevent RAW / WAR / WAW hazards

All threads must reach the barrier
- In conditional code, the condition must be uniform across the block
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
Coordinating Host & Device

- **Kernel launches are asynchronous**
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

- `cudaMemcpy()`: Blocks the CPU until the copy is complete
  - Copy begins when all preceding CUDA calls have completed

- `cudaMemcpyAsync()`: Asynchronous, does not block the CPU

- `cudaDeviceSynchronize()`: Blocks the CPU until all preceding CUDA calls have completed
Reporting Errors

All CUDA API calls return an error code (`cudaError_t`)
- Error in the API call itself
- OR
- Error in an earlier asynchronous operation (e.g. kernel)

Get the error code for the last error:
```
cudaError_t cudaGetLastError(void)
```

Get a string to describe the error:
```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```
Roadmap
Pascal GPU Features

NVLINK and Stacked Memory

NVLINK
- GPU high speed interconnect
- 80-200 GB/s

3D Stacked Memory
- 4x Higher Bandwidth (~1 TB/s)
- 3x Larger Capacity
- 4x More Energy Efficient per bit
NVLink
High-Speed GPU Interconnect
Questions?

Now you know CUDA C/C++. Go forth and accelerate!