Agenda

- Architecture
- Execution Model
- Memory Model
- Programming Model
- Challenges & Opportunities
Architecture
Tesla Unified Graphics and Computing Architecture
Scales parallel performance 2X beyond G80
240 Thread Processor cores, 30K threads
Double precision 64-bit IEEE 754 floating point
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
  - Variable 4-128 registers / thread

- 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 multithreaded
  - 32 SIMT warps of 32 threads
  - Independent thread execution
  - Mixed concurrent thread types

- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store
Software Architecture

Applications

- DX10
- OpenGL
- OpenCL
- CUDA C

Host OS

Device Driver

SIMT GPU

Hardware Thread Scheduling

- Hundreds of Cores
- Thousands of Threads

Scalar CPU
Execution Model
SIMT Multithreaded Execution

- **SIMT**: Single-Instruction Multi-Thread applies instruction to independent threads
- SIMT provides easy single-thread scalar programming with SIMD efficiency
- **Warp**: the set of 32 parallel threads that execute a SIMT instruction
- Hardware implements zero-overhead warp and thread scheduling
- Each thread processor manages state for up to 128 threads
- SIMT threads execute independently
- SIMT warp diverges and converges when threads branch independently
- Best efficiency and performance when threads of a warp execute together
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

- **Global Memory:** per-application
  - Shared by all threads
  - Inter-Grid communication

**Thread**
- Local Memory
- Register File

**Block**
- Shared Memory

**Grid 0**
- Sequential Grids in Time

**Grid 1**

**Global Memory**
Execution Model - Graphics

- Local Registers: per Pixel
  - Private per pixel
- 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

Pixel Warp

Pixel Grid 0

Pixel Grid 1
Memory Model
Hierarchical Memory

- Machines have hierarchical memory systems
  - It’s here to stay

- Energy consumption is dominated by moving data up and down (and across) this hierarchy

- Need efficient mechanisms for communication and synchronization

- Different machines have different hierarchies and portability is desired
Memory Model

- **Coherence**
  - Predominantly Software Managed
  - In many cases somewhat distributed

- **Consistence**
  - Hardware Managed per thread
  - Software Managed to outside world

- **Managed vs Expressed**
  - Memory (places)
  - Threads

- **Hardware/Software Cache Control**

<table>
<thead>
<tr>
<th></th>
<th>Managed Threads</th>
<th>Expressed Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managed Memory</td>
<td>Productivity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Programmer</td>
<td></td>
</tr>
<tr>
<td>Expressed Memory</td>
<td>Common</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Power Programmer</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2009 by NVIDIA Corporation

Workshop on Libraries and Autotuning for Petascale Applications 2009
Programming Model
CUDA Programming Model

- Based on C
  - Explicit about memory and its topology
  - Threads are managed (logical IDs)
  - Small language extensions to express device thread launches
  - Libraries for runtime functionality
    - Memory allocation, movement, synchronization, etc.

- Data Parallel Model
  - Write code ‘per thread’
  - Logic is aware of thread’s position in block and grid
  - Can share memory with other threads in same block
Grids, Blocks, and Threads

- Programmer partitions problem into a sequence of kernels
- A kernel executes as a grid of thread blocks
- A thread block is an array of threads that can cooperate
- Threads within the same block synchronize and share data in Shared Memory
- Execute thread blocks on multithreaded multiprocessor SM cores
Language Integration

- Declare C kernel functions and variables on GPU:
  ```c
  __global__ void KernelFunc(...);
  __device__ int GlobalVar;
  __shared__ int SharedVar;
  ```

- Call kernel function as Grid of 500 blocks with 128 threads per block:
  ```c
  KernelFunc<<< 500, 128 >>>(...);
  ```

- Explicit GPU memory allocation, CPU-GPU memory transfers
  - `cudaMalloc()`, `cudaFree()`
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...
Example: Add Arrays

### C program

```c
void addMatrix
  (float *a, float *b, float *c, int N)
{
  int i, j, idx;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      idx = i + j*N;
      c[idx] = a[idx] + b[idx];
    }
  }
}

void main()
{
  ......
  addMatrix(a, b, c, N);
}
```

### CUDA C program

```c
__global__ void addMatrixG
  (float *a, float *b, float *c, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int idx = i + j*N;
  if (i < N && j < N)
    c[idx] = a[idx] + b[idx];
}

void main()
{
  dim3 dimBlock (blocksize, blocksize);
  dim3 dimGrid (N/blockDim.x, N/blockDim.y);
  addMatrixG<<dimGrid, dimBlock>>>(a, b, c, N);
}
```
How to Scale GPU Computing?

- GPU parallelism scales widely
  - Ranges from 8 to many 100s of cores
  - Ranges from 100 to many 1000s of threads

- Graphics performance scales with GPU parallelism
  - Data parallel mapping of pixels to threads
  - Unlimited demand for parallel pixel shader threads and cores

Challenge:

- Scale **Computing** performance with GPU parallelism
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling
Challenges & Opportunities
Trends and Observations (1)

- Everything isn’t regular data parallel
  - Invest in exploiting irregular parallelism
  - Dynamic data driven execution

- Heterogeneous cores
  - More systems with single or selected purpose
  - CUDA/DP is nice but works best for homogenous

- Caches, Memory, Registers not necessarily getting larger
  - RC increases as mfg process decreases
  - Difficult to future proof against
Trends and Observations (2)

- Memory hierarchy will remain
  - Likely to become more complicated
  - Portability problems

- We need a convention to universally express memory/processor topology
  - Seems simpler than auto discovery
  - Let it be composable, please

- Need for dynamic tuning
  - Number of cores generally doubles per family
  - Low end has substantially less cores than high end
  - Ranges from 10s to 100s
Trends and Observations (3)

- Focus on load/store aspect of workloads
  - Programs tend to Load; Process; Store; Repeat;
  - Least portable part of the problem
  - Energy consumption is dominated by moving data through memory hierarchy
  - Is this a strength reduction opportunity? \( M+N << M*N \)

- Opportunities for SW managed caches and hybrid caches

- Composition is really hard
  - But seems more tenable with entirely resource managed systems
Summary

- Data Parallel Programming scales well
  - But you still have to do all the work!
  - Complicated by varying memory topologies
  - Complicated by heterogeneous systems

- Autotuning can address
  - Efficient data propagation to/from cores (memory to cores)
  - Efficient core utilization (best core for job, threads to memory)
  - In field adaptation to new topologies

- We know we’re there when we only program to managed memory & cores
The End

Ben de Waal
ben@nvidia.com