Outline

Part 1 – CUDA Best Practice Strategies
- Assess
- Parallelize
- Optimize
- Deploy

Part 2 – CUDA Optimization
- Hardware
- Occupancy
- Divergence
- Bottlenecks
- Atomic Ops

Talk Time
- 10 min
- 20 min
- 30 min
- 40 min
- 50 min

Difficulty
- state change
Part #1 - Best Practices: Strategies
APOD: A Systematic Path to Performance

- Assess
- Parallelize
- Optimize
- Deploy
Assess

- Know your application problem
- Know your hardware capabilities
- Determine what aspects of problem are best suited to parallelization. Identify “hotspots”
- Use profiling tools for find critical bottlenecks in CPU code
Profiling and Debugging Solutions

- NVIDIA Nsight
  Eclipse & Visual Studio Editions

- NVIDIA CUDA-GDB
  for Linux & Mac

- NVIDIA CUDA-MEMCHECK
  for Linux & Mac

- Allinea DDT with CUDA
  Distributed Debugging Tool

- TotalView for CUDA
  for Linux Clusters

http://developer.nvidia.com/nsight
Assess

1. Know your hardware!

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>Gflops</th>
<th>MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core i7-3770</td>
<td>4</td>
<td>108</td>
<td>25</td>
</tr>
<tr>
<td>GeForce GTX480</td>
<td>480</td>
<td>1345</td>
<td>177</td>
</tr>
<tr>
<td>Quadro K5000</td>
<td>1536</td>
<td>2168</td>
<td>172</td>
</tr>
<tr>
<td>Tesla K20X</td>
<td>2688</td>
<td>3950</td>
<td>250</td>
</tr>
</tbody>
</table>
Assess

Practical Example: Fluid Simulation
Assess

2. Know your problem!

- Insert into Accel Grid
- Compute Forces
- Compute Pressures
- Integrate
Assess

2. Know your problem!

- Insert into Accel Grid
- Compute Forces
- Compute Pressures
- Integrate

Search for neighboring particles. (NNS)

Like to be slowest part of code.

CPU version:
\[ O(n^2) \] worst case
\[ O(nk) \] spatial grid lookup
Assess

3. Determine metrics

Time:
Standardize your units (avoid using fps)
Consider time to complete task, time per frame, and time per sub-task.
e.g. milliseconds

Performance:
Measures the overall ability to do work.
Choose a reasonable metric.. e.g. Image processing.. pixels/sec
Combination of algorithm efficiency and hardware.
e.g. particles / second  ==  particles / op * ops / second

Efficiency:
Normalizes performance by dividing by hardware Gflops.
Measures the capability of the algorithm regardless of hardware.
e.g. (particles / second) / Gflops  ==  particles / Gflop
Assess

4. Identify hotspots

524,288 particles

Total: 1300 ms / frame  
Power = 403,289 particles / sec  
Efficiency = 186 p/s/Gf
Assess

4. Identify hotspots

524,288 particles

- Insert: 7 ms
- Pressure: 480 ms
- Force: 788 ms
- Advance: 36 ms

Order of magnitude greater than other steps
Parallelize

• Determine amount of *crosstalk* in the problem
• Identify parallel method suitable to problem
• Translate CPU algorithms to GPU
Parallelize

1. Crosstalk and Coherency determine ease of parallelism

Color Grading
Simple Particles
Image Blur
N-Body Problem
Fluid Simulation
Raytracing

coherent

incoherent
Parallelize

2. Design parallel algorithm

Example: Fluid Simulation

Key Observations

1. Particles are dynamic

2. Particles become incoherent in memory (mix) as they move

3. Radix-Sort can keep coherency. Radix = fast parallel sort.

4. Do Neighbor Search on coherent particles.

Assign one particle per thread. Keep coherent by sorting each frame.

Many resources available: CUDA SDK Samples developer.nvidia.com/gpu-computing-sdk Developer Forums devtalk.nvidia.com
Optimize

1. Compare GPU to CPU

CPU Time: 1300 ms
CPU Pow: 403,289 p/sec
CPU Effic: 3734 p/s/Gf

GPU Time: 90 ms / frame
GPU Pow: 5,825,422 p/sec
GPU Effic: 2687 p/s/Gf

14x faster
Optimize

2. Memory Architecture

Global Memory
(inc. Local Memory)
170 GB/s
(400 cyl.)

L2 Cache, 1.5MB GK110
2000 GB/s

Shared Memory, 64k
(shared per SMX)
Read-only, 48k

Texture Memory
L1 Cache
(100 cyl.)

Registers
8000 GB/s

Kepler Memory Hierarchy
3. Keep Optimizing!

What is occupancy?

Why is it 56% for Forces?

Why is shared memory not used?

Shared mem: 100x faster
Deploy

Once you have a working, efficient GPU solution…

- **Multiple GPUs**: cudaGetDeviceCount
- **Error handling**: cudaGetErrorString()
- **NVML**: Cluster management
- **NV-SMI**: System monitoring
Part #2 - Best Practices: CUDA Optimization
Hardware Architecture

SimpleGPU

A visual simplification of the GPU with all the essential components, to help visualize optimization issues.
Hardware Architecture

Fermi
GF100
GF104

Kepler
GK104
GK110

Global Memory
GTX 480 = 1.5G
Titan / K20 = 6 GB

Local Memory
variable (uses GMEM)

Shared Memory
48k 48k 48k 48k

Registers / Thread
63 63 63 255

Cores / MP
32 32 192 192

Threads / Warp
32 32 32 32

Threads / Threadblock
1024 1024 1024 1024

Know your hardware.
**Execution Model**

Threads = virtual, millions

Cores = limited, physical

Many threads (virtual) are scheduled to run on cores (physical hardware)

**SMX** = Streaming multi-processors
- Run a threadblock (multiple warps)
- Shares shared memory
- Provides registers to each thread

All threads in a warp are launched in parallel. Instructions and memory reads are executed in parallel within a warp.
Occupancy

1. Maximize use of all SMX on the GPU
2. Maximize use of threads in a warp per SMX
Occupancy #1 - Maximize GPU Usage

C Code:
```c
for (i = 0; i < 1024; i++)
    y = data[i] ^ 2;
```

CUDA Code:
```c
kernel < grid, tblk > ( data )
{
    int i = threadIdx.x;
    int y = data[i] ^ 2;
}
```
Occupancy #1 - Maximize GPU Usage

Dim2 tblk (16, 16) = 256 threads
Dim1 grid (1, 1)

kernel < grid, tblk > (my_img, grey)

“Hey, great, 256 steps in parallel”
Occupancy #1 - Maximize GPU Usage

Dim2 tblk (16, 16)
Dim2 grid (2, 1) ➔ 2x work

kernel < grid, tblk > (my_img, grey)

“Wow, double the calculations!”

It takes the same amount of time!
Most of the GPU is just sitting there.
Occupancy #1 - Maximize GPU Usage

Now we’re doing ~1,024 in parallel!

... AND giving GPU enough to stay busy.

Total: 1,048,576 threads scheduled

#1. Maximize use of SMX in GPU
Irregular threadblock dimensions cause low occupancy in each warp, and tail threads….
even though the grid is large.

3 * 10 = 30 (not 32) >> non-full warp
3 * 25 = 75 (not 96) >> non-full threadblock (tails)
Occupancy #2 - Threadblocks

Dim2 tblk (16, 1)
Dim2 grid (64, 64)
kernel < grid, tblk > (my_img, grey)

Only 16 threads per threadblock.
GPU supports:
- 32 threads / warp
- 1024 threads / threadblock

Dim2 tblk (32, 32)
Dim2 grid (64, 64)
kernel < grid, tblk > (my_img, grey)

Now: 1024 threads / threadblock
Now threadblocks are full.

#1. Maximize use of threadblocks
Execution Divergence

1. Reduce or eliminate the use of conditionals

2. Maximize computational ops
   (over conditional ops and memory ops)
Execution Divergence

```
kernel < grid, tblk > ( in, out, param )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if ( in[ i+1 ] > 0 ) {
        out[ i ] = pow ( in[ i ], in[ i+1 ] );
    } else {
        out[ i ] = 1;
    }
}
```

Code makes sure value is in range. Not an issue across SMX.

Warp #1

Warp #2 – *must wait for all to finish*
Execution Divergence

```
kernel < grid, tblk > ( in, out, param )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    out[ i ] = pow ( in[ i ], in[ i+1 ] );
}
```

Do validation on input data before launching kernel.

Warp #1

Warp #2 – next warp launches sooner
kernel < grid, tblk > ( in, out, param )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    val = in [ i ];
    if ( isEnabled[0] ) val = pow ( val, param[0] );
    if ( isEnabled[1] ) val = val * param[1];
    if ( isEnabled[2] ) val = val + param[2];
    out [ i ] = val;
}

Image processing: Conditionally perform a power, multiply and offset on each pixel.
Execution Divergence

kernel < grid, tblk > ( in, out, param )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    val = in [ i ];
    if ( isEnabled[0] ) val = pow ( val, param[0] );
    if ( isEnabled[1] ) val = val * param[1];
    if ( isEnabled[2] ) val = val + param[2];
    out [ i ] = val;
}

Image processing: Conditionally perform a power, multiply and offset on each pixel.

Most of the time is spent loading and checking parameters, rather than computing.

Warp #1

Warp #2 – must wait for all to finish
Execution Divergence

```
kernel < grid, blk > ( in, out, param )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    out[i] = pow ( in[i], param[0] ) * param[1] + param[2];
}
```

Solution: Use parameters to set identity values when not enabled. e.g. `val = val * 1;  val = val + 0;`

- #1. Avoided global load of isEnabled
- #2. Avoided conditional checks
- #3. Avoided register use (val = )

Maximize use of the GPU for calculation. Memory reads and conditionals are not always “useful” work.
Bottleneck: Memory or Instruction?

1. Are you memory or instruction-limited?
2. If memory-limited, seek to reduce bandwidth.
3. If instruction-limited, seek to remove unnecessary calculations.
Theoretical Limits

MEMORY

Theoretical *memory* throughput: 177 GB/s (Tesla M2090)

INSTRUCTIONS

Refer to CUDA Programming Guide for ops/cycle

Theoretical *instruction* throughput: 655 Ginst/sec (Tesla M2090)

BALANCED RATIO

This is what we want to optimize: 3.76 : 1

higher ratio = instruction-bound
lower ratio = memory-bound
kernel < 3, 25, 1 > ( my_img, grey )
{
    int px = threadIdx.y * blockIdx.x + threadIdx.x
    int a = img[px-1];
    int b = img[px];    // filter operation
    int c = img[px+1];
    int d;
    d = a + b + c;      // combine pixel values
}

LOADS: 4x 32-bit global per thread
        = 16 bytes / thread

OPS:  3x adds per thread

Ratio = 16:3 = 5.3:1  great than 4:1  memory bound
Memory Limited - Soln #1. Reduce data size

```c
int px = threadIdx.y * blockIdx.x + threadIdx.x
int a = img[px-1];
int b = img[px];
int c = img[px+1];
int d;
d = a + b + c;
```

Kernel < 3, 25, 1> (my_img, grey)

- LOADS: 4x 8-bit global per thread
  = 4 bytes / thread
- OPS: 3x add per thread
- Ratio = 4:3 = 1.3:1  
  *less than 4:1*
  
  no longer memory bound
kernel < 3, 25, 1 > ( my_img, grey )
{
    int i = threadIdx.y * blockIdx.x + threadIdx.x
    __shared__ pixels[32];
    if ( threadIdx.x < 32 )
        pixels[ threadIdx.x ] = img[ i ];
    __syncthreads();
    tx = threadIdx.x;
    img[ i ] = pixels[tx-1] + pixels[tx] + pixels[tx+1];
}

Share memory is 100x faster than global.

LOADS: 1 bytes per thread (each does 1)
OPS: 3x add per thread
Ratio = 1:3 = 0.3:1 much less than 4:1
not memory bound
Coalescing and Memory Access

Scenario:
- Warp requests 32 aligned, *consecutive* 4-byte words
- Warp needs 128 bytes - moved together across bus
Coalescing and Memory Access

Scenario:
- Warp requests 32 aligned, *permuted* 4-byte words
- Perfectly coalesced

Addresses from a warp:

Memory addresses:
0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
Scenario:
- Warp requests 32 misaligned, consecutive 4-byte words
- At most 160 bytes move across the bus. No coalesced
Scenario:
- All threads in a warp request the same 4-byte word
Coalescing and Memory Access

Scenario:
- Warp requests 32 scattered 4-byte words, outside 128 bytes
- Non-coalesced
Instruction Limited

Assuming that data is loaded efficiently (not memory-bound):

#1. Are you performing unnecessary calculations?

#2. Can the problem/algorithn be reformulated?

#3. Use faster ops: Can you use float instead of double?

Generally, a good problem to have:

It means most of your effort is going to do *useful work*.
- Memory ops are necessary, but not useful
- Conditional may be necessary, but often are not
Instruction Limited

 kernel < 3, 25, 1 > ( my_img, grey )
 {
    int px = threadIdx.y * blockIdx.x + threadIdx.x
    int a = img[px];
    int d;

    d = pow(a, 3.2) * sin(a) + a*a;

    LOADS: 1x 4-bit global LOADS per thread
          = 4 bytes / thread

    OPS: 3x add/mul + (2x pow/sin * 6) = 15

    Ratio = 1:15 = 1:15 much less than 4:1
    Instruction bound

CUDA Programming Guide (Section 5.4.1)
kernel < 3, 25, 1 > ( my_img, grey )
{
    int px = threadIdx.y * blockIdx.x + threadIdx.x

    int a = img[px];
    int d;

    d = pow(a, 3.2) * sin(a) + a*a;
}

Are your math operations optimal?

Common approaches:
- Avoid unnecessary calculations
- CUDA fast, half-precision ops.
- Lookup tables, constant memory
- Algorithmic…
Assess

Deploy

Parallelize

Optimize

Know your hardware

Find hotspots
Assess

Parallelize

Optimize

Deploy

Know your hardware

Find hotspots

Write CUDA code

Profile CPU and GPU
Assess

Parallelize

Optimize

Deploy

Know your hardware

Find hotspots

Write CUDA code

Profile CPU and GPU

Coalescing

Branching

Bandwidth

Occupancy
Assess
Parallelize
Optimize
Deploy

Know your hardware
Find hotspots
Write CUDA code
Profile CPU and GPU

Distribute!

Multiple Devices
Error Handling
Coalescing
Branching
Bandwidth
Occupancy
Thank You!

Further Resources:

CUDA SDK Samples:  https://developer.nvidia.com/gpu-computing-sdk
GPU Technology Conference (GTC) - Featured talks:  http://www.gputechconf.com

RAMA HOETZLEIN  rhoetzlein@nvidia.com