The art of doing more with less
RULE #1: DON’T TRY TOO HARD
RULE #1: DON’T TRY TOO HARD

Performance

Unrealistic Effort/Reward

Peak Performance

Time
RULE #1: DON’T TRY TOO HARD
RULE #1: DON’T TRY TOO HARD

- Reduce this time
- Don’t waste this time
- Get on this curve
RULE #1: DON’T TRY TOO HARD

Performance

Time

Peak Performance

Premature excitement

Point of diminishing returns

Trough of despair

Wait, it’s going slower??

4 weeks and this is it?

Most people give up here

Actual Effort/Reward

Here be ninjas

Hire an intern
PERFORMANCE CONSTRAINTS

- Memory: 75%
- Occupancy: 10%
- Instruction: 2%
- Divergence: 3%
- Compute Intensity: 10%
PERFORMANCE CONSTRAINTS

Chart Title

- Compute Intensity
- Divergence
- Instruction
- Occupancy
- Divergent Access
- Cache Inefficiency
- Register Spilling
- Coalescence
- CPU <> GPU Transfer
MEMORY ORDERS OF MAGNITUDE

<table>
<thead>
<tr>
<th>SM</th>
<th>L1$</th>
<th>L2 Cache</th>
<th>GDRAM</th>
<th>DRAM</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>regs</td>
<td>shmem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regs</td>
<td>shmem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regs</td>
<td>shmem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- L1$: 150 GB/sec
- L2: 16 GB/sec
- GDRAM: 300 GB/sec
- DRAM: 2,000 GB/sec
- CPU: 20,000 GB/sec
- PCIe bus: 150 GB/sec
TALK BREAKDOWN

In no particular order

1. Why Didn’t I Think Of That?
2. CPU Memory to GPU Memory (the PCIe Bus)
3. GPU Memory to the SM
4. Registers & Shared Memory
5. Occupancy, Divergence & Latency
6. Weird Things You Never Thought Of (and probably shouldn’t try)
WHERE TO BEGIN?
THE OBVIOUS

Start with the Visual Profiler
CPU <> GPU DATA MOVEMENT
PCI ISSUES

Moving data over the PCIe bus

16 GB/sec
PIN YOUR CPU MEMORY
PIN YOUR CPU MEMORY
PIN YOUR CPU MEMORY
PIN YOUR CPU MEMORY

CPU allocates & pins page then copies locally before DMA
PIN YOUR CPU MEMORY

cudaHostAlloc( &data, size, cudaHostAllocMapped );
cudaHostRegister( &data, size, cudaHostRegisterDefault );
PIN YOUR CPU MEMORY

Host<>Device Data Movement Performance
(PCle gen-2)
REMEMBER: PCIe GOES BOTH WAYS
Operations in a single stream are ordered

But hardware can copy and compute at the same time

STREAMS & CONCURRENCY

Hiding the cost of data transfer
STREAMS & CONCURRENCY

Stream 2

Copy up  Work  Copy back

Stream 1

Copy up  Work  Copy back

Single Stream

Copy data to GPU  Compute  Copy data to Host

Saved Time
STREAMS & CONCURRENCY

Can keep on breaking work into smaller chunks and saving time
SMALL PCIe TRANSFERS

PCle is designed for large data transfers
But fine-grained copy/compute overlap prefers small transfers
So how small can we go?
APPARENTLY NOT THAT SMALL

PCle (Gen3) Data Transfer Rate by Copy Size
Blue line is single-element burst, red line is large copy done in chunks

Copy Bandwidth MB/sec

Copy Size
FROM GPU MEMORY TO GPU THREADS
FEEDING THE MACHINE

From GPU Memory to the SMs
USE THE PARALLEL ARCHITECTURE

Hardware is optimized to use all SIMT threads at once

- Threads run in groups of 32
- Cache is sized to service sets of 32 requests at a time
- High-speed GPU memory works best with linear access
VECTORIZE MEMORY LOADS

Multi-Word as well as Multi-Thread
VECTORIZE MEMORY LOADS

Fill multiple cache lines in a single fetch
VECTORIZE MEMORY LOADS

Fill multiple cache lines in a single fetch
VECTORIZE MEMORY LOADS

Data Movement Rate vs. Data Size
(1 element per thread)

GB/sec

0 5 10 15 20 25 30 35 40

MB Copied

0 20 40 60 80 100 120

4-byte
8-byte
16-byte
DO MULTIPLE LOADS PER THREAD
Multi-Thread, Multi-Word AND Multi-Iteration

__global__ void copy(int2 *input, int2 *output, int max) {
    int id = threadIdx.x + blockDim.x * blockIdx.x;
    if (id < max) {
        output[id] = input[id];
    }
}

__global__ void copy(int2 *input, int2 *output, int max, int loadsPerThread) {
    int id = threadIdx.x + blockDim.x * blockIdx.x;
    for (int n=0; n<loadsPerThread; n++) {
        if (id >= max) {
            break;
        }
        output[id] = input[id];
        id += blockDim.x * gridDim.x;
    }
}

One copy per thread
Maximum overhead

Multiple copies per thread
Amortize overhead
"MAXIMAL" LAUNCHES ARE BEST
COALESCED MEMORY ACCESS
It’s not just good enough to use all SIMT threads

Coalesced: Sequential memory accesses are adjacent

Uncoalesced: Sequential memory accesses are unassociated
SIMT PENALTIES WHEN NOT COALESCED

Single 32-wide operation

32 one-wide operations
SCATTER & GATHER

Gathering

Read sequentially
Write sequentially

Scattering

Read randomly
Write randomly
AVOID SCATTER/GATHER IF YOU CAN

Scatter/Gather Behaviour
(4-byte data, maximal grid launch)

- direct
- linear
- scatter
- gather
- both
AVOID SCATTER/GATHER IF YOU CAN
SORTING MIGHT BE AN OPTION

If reading non-sequential data is expensive, is it worth sorting it to make it sequential?
SORTING MIGHT BE AN OPTION

Even if you’re only going to read it twice, then yes!
PRE-SORTING TURNS OUT TO BE GOOD
DATA LAYOUT: “AOS vs. SOA”
Sometimes you can’t just sort your data

Array-of-Structures

#define NPTS 1024 * 1024

struct Coefficients_AOS {
    double u[3];
    double x[3][3];
    double p;
    double rho;
    double eta;
};

Coefficients_AOS gridData[NPTS];

Single-thread code prefers arrays of structures, for cache efficiency

Structure-of-Arrays

#define NPTS 1024 * 1024

struct Coefficients_SOA {
    double u[3][NPTS];
    double x[3][3][NPTS];
    double p[NPTS];
    double rho[NPTS];
    double eta[NPTS];
};

Coefficients_SOA gridData;

SIMT code prefers structures of arrays, for execution & memory efficiency
DATA LAYOUT: “AOS vs. SOA”

#define NPTS 1024 * 1024

struct Coefficients_AOS {
    double u[3];
    double x[3][3];
    double p;
    double rho;
    double eta;
};

Coefficients_AOS gridData[NPTS];

Structure Definition

Conceptual Layout
SOA: STRIDED ARRAY ACCESS

GPU reads data one element at a time, but in parallel by 32 threads in a warp

double u0 = gridData[threadIdx.x].u[0];
AOS: COALESCED BUT COMPLEX

GPU reads data one element at a time, but in parallel by 32 threads in a warp

double u0 = gridData.u[0][threadIdx.x];
BLOCK-WIDE LOAD VIA SHARED MEMORY

Read data linearly as bytes. Use shared memory to convert to struct

Block copies data to shared memory

Device Memory

Shared Memory
BLOCK-WIDE LOAD VIA SHARED MEMORY

Read data linearly as bytes. Use shared memory to convert to struct
CLEVER AOS/SOA TRICKS

Speedup of SOA conversion over raw AOS

Structure size in bytes

Speedup of SOA vs. AOS

int4
CLEVER AOS/SOA TRICKS

Helps for any data size
HANDY LIBRARY TO HELP YOU

Trove - A utility library for fast AOS/SOA access and transposition

https://github.com/bryancatanzaro/trove

Random AoS Access

Contiguous AoS Access

GB/s

Size of structure in bytes

GB/s

Size of structure in bytes
(AB)USING THE CACHE
L2 cache is fast but small:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>L2 Cache Size</th>
<th>Total Threads</th>
<th>Cache Bytes per Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kepler</td>
<td>1536 KB</td>
<td>30,720</td>
<td>51</td>
</tr>
<tr>
<td>Maxwell</td>
<td>3072 KB</td>
<td>49,152</td>
<td>64</td>
</tr>
<tr>
<td>Pascal</td>
<td>4096 KB</td>
<td>114,688</td>
<td>36</td>
</tr>
</tbody>
</table>

L2 Cache

<table>
<thead>
<tr>
<th></th>
<th>GDRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,000 GB/sec</td>
<td>300 GB/sec</td>
</tr>
</tbody>
</table>

MAKING THE MOST OF L2-CACHE
TRAINING DEEP NEURAL NETWORKS
LOTS OF PASSES OVER DATA

fft

W1  3x3 convolution
W2  5x5 convolution
W3  7x7 convolution

+  

Cat!
MULTI-RESOLUTION CONVOLUTIONS

Pass 1: 3x3
Pass 2: 5x5
Pass 3: 7x7
TILED, MULTI-RESOLUTION CONVOLUTION

Do 3 passes per-tile

Each tile sized to fit in L2 cache
LAUNCHING FEWER THAN MAXIMUM THREADS
SHARED MEMORY: DEFINITELY WORTH IT
USING SHARED MEMORY WISELY

Shared memory arranged into “banks” for concurrent SIMT access
  ▪ 32 threads can read simultaneously so long as into separate banks

Shared memory has 4-byte and 8-byte “bank” sizes
Many algorithms have high data re-use: potentially good for shared memory

“Stencil” algorithms accumulate data from neighbours onto a central point
  - Stencil has width “W” (in the above case, W=5)

Adjacent threads will share (W-1) items of data - good potential for data re-use
STENCILS IN SHARED MEMORY

Impact of Data Size on Stencil Operation
(8-byte banks, maximal grid launch)

Speedup Shared vs. Global

Stencil Size
SIZE MATTERS

Shared- vs. Global-Memory Stencil Operations, Compared by Block Size
PERSISTENT KERNELS
Revisiting the tiled convolutions

Avoid multiple kernel launches by caching in shared memory instead of L2

```c
void tiledConvolution() {
    convolution<3><<< numblocks, blockdim, 0, s >>>(ptr, chunkSize);
    convolution<5><<< numblocks, blockdim, 0, s >>>(ptr, chunkSize);
    convolution<7><<< numblocks, blockdim, 0, s >>>(ptr, chunkSize);
}
```

Separate kernel launches with L2 re-use

```c
__global__ void convolutionShared(int *data, int count, int sharedelems) {
    extern __shared__ int shdata[];
    shdata[threadIdx.x] = data[threadIdx.x + blockDim.x*blockIdx.x];
    __syncthreads();
    convolve<3>(threadIdx.x, shdata, sharedelems);
    __syncthreads();
    convolve<5>(threadIdx.x, shdata, sharedelems);
    __syncthreads();
    convolve<7>(threadIdx.x, shdata, sharedelems);
}
```

Single kernel launch with persistent kernel
PERSISTENT KERNELS

Tiled Multi-Convolution Performance

GB/sec data processed

Tile size in MB

L2
Shared
Can save memory copies. It’s obvious when you think about it …

Compute only begins when 1\textsuperscript{st} copy has finished. Task only ends when 2\textsuperscript{nd} copy has finished.

Compute begins after first fetch. Uses lots of threads to cover host-memory access latency. Takes advantage of bi-directional PCI.
OPERATING DIRECTLY FROM CPU MEMORY
OCCUPANCY AND REGISTER LIMITATIONS

Register file is bigger than shared memory and L1 cache!

Occupancy can kill you if you use too many registers

Often worth forcing fewer registers to allow more blocks per SM

But watch out for math functions!

```
__launch_bounds__(maxThreadsPerBlock,
                  minBlocksPerMultiprocessor)
__global__ void compute() {
    y = acos(pow(log(fdivide(tan(cosh(erfc(x))), 2)), 3));
}
```
THANK YOU!

NVIDIA.