S7444 - WHAT THE PROFILER IS TELLING YOU: OPTIMIZING GPU KERNELS

Christoph Angerer, Jakob Progsch,
GTC 2017
BEFORE YOU START
The five steps to enlightenment

1. Know your application
   • What does it compute? How is it parallelized? What final performance is expected?

2. Know your hardware
   • What are the target machines, how many nodes? Machine-specific optimizations okay?

3. Know your tools
   • Strengths and weaknesses of each tool? Learn how to use them (and learn one well!)

4. Know your process
   • Performance optimization is a constant learning process

5. Make it so!
THE APOD CYCLE

1. Assess
   - Identify Performance Limiter
   - Analyze Profile
   - Find Indicators

2. Parallelize

3. Optimize
   - 3b. Build Knowledge

4. Deploy and Test
GUIDING OPTIMIZATION EFFORT
“Drilling Down into the Metrics”

• **Challenge:** How to know where to start?

• **Top-down Approach:**
  - Find Hotspot Kernel
  - Identify Performance Limiter of the Hotspot
  - Find performance bottleneck indicators related to the limiter
  - Identify associated regions in the source code
  - Come up with strategy to fix and change the code
  - Start again
KNOW YOUR APPLICATION: HPGMG
HPGMG
High-Performance Geometric Multi-Grid, Hybrid Implementation

Fine levels are executed on throughput-optimized processors (GPU)
Coarse levels are executed on latency-optimized processors (CPU)

http://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg/
MULTI-GRID BOTTLENECK

Cost of operations

MOST TIME SPENT ON STENCILS
KNOW YOUR HARDWARE:
PASCAL ARCHITECTURE
# GPU COMPARISON

<table>
<thead>
<tr>
<th></th>
<th>P100 (SXM2)</th>
<th>M40</th>
<th>K40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double/Single/Half TFlop/s</td>
<td>5.3/10.6/21.2</td>
<td>0.2/7.0/NA</td>
<td>1.4/4.3/NA</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/s)</td>
<td>732</td>
<td>288</td>
<td>288</td>
</tr>
<tr>
<td>Memory Size</td>
<td>16GB</td>
<td>12GB, 24GB</td>
<td>12GB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>4096 KB</td>
<td>3072 KB</td>
<td>1536 KB</td>
</tr>
<tr>
<td>Base/Boost Clock (Mhz)</td>
<td>1328/1480</td>
<td>948/1114</td>
<td>745/875</td>
</tr>
<tr>
<td>TDP (Watts)</td>
<td>300</td>
<td>250</td>
<td>235</td>
</tr>
</tbody>
</table>
# GP100 SM

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>64</td>
</tr>
<tr>
<td>Register File</td>
<td>256 KB</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>64 KB</td>
</tr>
<tr>
<td>Active Threads</td>
<td>2048</td>
</tr>
<tr>
<td>Active Blocks</td>
<td>32</td>
</tr>
</tbody>
</table>
KNOW YOUR TOOLS: PROFILERS
PROFILING TOOLS

Many Options!

From NVIDIA

- nvprof
- NVIDIA Visual Profiler
  - Standalone (nvvp)
  - Integrated into Nsight Eclipse Edition (nsight)
- Nsight Visual Studio Edition

Third Party

- TAU Performance System
- VampirTrace
- PAPI CUDA component
- HPC Toolkit
  - (Tools using CUPTI)

Without loss of generality, in this talk we will be showing nvvp screenshots
THE NVVP PROFILER WINDOW

• S7824 – DEVELOPER TOOLS UPDATE, Wed 4:00 PM
• S7495 - OPTIMIZING APPLICATION PERFORMANCE WITH CUDA PROFILING TOOLS, Thur 10:00 AM
MAKE IT SO: ITERATION 1

2ND ORDER 7-POINT STENCIL
IDENTIFY HOTSPOT

Identify the hotspot: `smooth_kernel()`

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Version</td>
<td>0.109443s</td>
<td>1.00x</td>
</tr>
</tbody>
</table>
IDENTIFY PERFORMANCE LIMITER

Results

Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla P100-PCIE-16GB". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Memory Ops

Load/Store

Memory Utilization Issues?
PERFORMANCE LIMITER CATEGORIES

Memory Utilization vs Compute Utilization

Four possible combinations:

- Compute Bound
- Bandwidth Bound
- Latency Bound
- Compute and Bandwidth Bound
GPU Utilization May Be Limited By Block Size

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.
OCCUPANCY
GPU Utilization

Each SM has limited resources:

- max. 64K Registers (32 bit) distributed between threads
- max. 48KB of shared memory per block (96KB per SMM)
- max. 32 Active Blocks per SMM
- Full occupancy: 2048 threads per SM (64 warps)

When a resource is used up, occupancy is reduced

(*) Values vary with Compute Capability
GPUs cover latencies by having a lot of work in flight

- The warp issues
- The warp waits (latency)

**Fully covered latency**
- Warp 0
- Warp 1
- Warp 2
- Warp 3
- Warp 4
- Warp 5
- Warp 6
- Warp 7
- Warp 8
- Warp 9

**Exposed latency, not enough warps**
- Warp 0
- Warp 1
- Warp 2
- Warp 3

No warp issues
LATENCY AT HIGH OCCUPANCY

Many active warps but with high latency instructions

Exposed latency at high occupancy

warp 0
warp 1
warp 2
warp 3
warp 4
warp 5
warp 6
warp 7
warp 8
warp 9

No warp issuing
LOOKING FOR MORE INDICATORS

Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern. For each load or store, improve the alignment and access pattern of the memory access.

Line / File: smooth_base.h \home\cangere\projects\GTC2017\hpreg-cuda\finite-volune\source\cuda\stencils

87 Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [ 6291456 L2 transactions for 524288 total executions]
87 Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [ 6291456 L2 transactions for 524288 total executions]
87 Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [ 6291456 L2 transactions for 524288 total executions]
87 Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [ 6291456 L2 transactions for 524288 total executions]

For line numbers use: nvcc -lineinfo

12 Global Load Transactions per 1 Request
MEMORY TRANSACTIONS: BEST CASE

A warp issues 32x4B aligned and consecutive load/store request

Threads read different elements of the same 128B segment

1x 128B load/store request per warp

1x 128B L1 transaction per warp

4x 32B L2 transactions per warp

1x L1 transaction: 128B needed / 128B transferred

4x L2 transactions: 128B needed / 128B transferred
MEMORY TRANSACTIONS: WORST CASE

Threads in a warp read/write 4B words, 128B between words

Each thread reads the first 4B of a 128B segment

Stride: 32x4B

1x 128B load/store request per warp

1x 128B L1 transaction per thread

1x 32B L2 transaction per thread

32x L1 transactions: 128B needed / 32x 128B transferred

32x L2 transactions: 128B needed / 32x 32B transferred
TRANSACTIONS AND REPLAYS

With replays, requests take more time and use more resources

More instructions issued

More memory traffic

Increased execution time
FIX: BETTER GPU TILING

### Before
- Grid Size: [65536,1,1]
- Block Size: [8,4,1]

### After
- Grid Size: [16384,1,1]
- Block Size: [32,4,1]

#### Block Size Up

#### Transactions Per Access Down

#### Memory Utilization Up

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Version</td>
<td>0.109443s</td>
<td>1.00x</td>
</tr>
<tr>
<td>Better Memory Accesses</td>
<td>0.076051s</td>
<td>1.44x</td>
</tr>
</tbody>
</table>
**PERF-OPT QUICK REFERENCE CARD**

<table>
<thead>
<tr>
<th>Category:</th>
<th>Latency Bound - Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Latency is exposed due to low occupancy</td>
</tr>
<tr>
<td>Goal:</td>
<td><strong>Hide</strong> latency behind more parallel work</td>
</tr>
<tr>
<td>Indicators:</td>
<td>Occupancy low (&lt; 60%)</td>
</tr>
<tr>
<td></td>
<td>Execution Dependency High</td>
</tr>
<tr>
<td>Strategy:</td>
<td>Increase occupancy by:</td>
</tr>
<tr>
<td></td>
<td>• Varying block size</td>
</tr>
<tr>
<td></td>
<td>• Varying shared memory usage</td>
</tr>
<tr>
<td></td>
<td>• Varying register count (use __launch_bounds)</td>
</tr>
</tbody>
</table>
**PERF-OPT QUICK REFERENCE CARD**

<table>
<thead>
<tr>
<th>Category:</th>
<th>Latency Bound - Coalescing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Memory is accessed inefficiently =&gt; high latency</td>
</tr>
<tr>
<td>Goal:</td>
<td>Reduce #transactions/request to reduce latency</td>
</tr>
<tr>
<td>Indicators:</td>
<td>Low global load/store efficiency, High #transactions/#request compared to ideal</td>
</tr>
</tbody>
</table>
| Strategy: | Improve memory coalescing by:  
  • Cooperative loading inside a block  
  • Change block layout  
  • Aligning data  
  • Changing data layout to improve locality |
<table>
<thead>
<tr>
<th>Category:</th>
<th>Bandwidth Bound - Coalescing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Too much unused data clogging memory system</td>
</tr>
<tr>
<td>Goal:</td>
<td>Reduce traffic, move more <strong>useful</strong> data per request</td>
</tr>
<tr>
<td>Indicators:</td>
<td>Low global load/store efficiency,</td>
</tr>
<tr>
<td></td>
<td>High transactions/#request compared to ideal</td>
</tr>
<tr>
<td>Strategy:</td>
<td>Improve memory coalescing by:</td>
</tr>
<tr>
<td></td>
<td>• Cooperative loading inside a block</td>
</tr>
<tr>
<td></td>
<td>• Change block layout</td>
</tr>
<tr>
<td></td>
<td>• Aligning data</td>
</tr>
<tr>
<td></td>
<td>• Changing data layout to improve locality</td>
</tr>
</tbody>
</table>
ITERATION 2:
REGISTER OPTIMIZATION AND CACHING
NEW PERFORMANCE LIMITER: MEMORY BANDWIDTH

Kernel Performance Is Bound By Memory Bandwidth
For device "Tesla P100-PCIE-16GB" the kernel’s compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.

L2 Cache

<table>
<thead>
<tr>
<th></th>
<th>Reads</th>
<th>635.639 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Writes</td>
<td>55.772 GB/s</td>
</tr>
<tr>
<td>Total</td>
<td>62393208</td>
<td>691.411 GB/s</td>
</tr>
</tbody>
</table>

Device Memory

<table>
<thead>
<tr>
<th></th>
<th>Reads</th>
<th>440.087 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Writes</td>
<td>48.867 GB/s</td>
</tr>
<tr>
<td>Total</td>
<td>44123412</td>
<td>488.954 GB/s</td>
</tr>
</tbody>
</table>
GPU MEMORY HIERARCHY

P100 (SMX2)

- Registers (256 KB/SM): good for intra-thread data reuse
- Shared memory (64 KB/SM): good for explicit intra-block data reuse
- L1$/$Tex$, L2$ (4096 KB): implicit data reuse

Bring reused data closer to the SMs
STENCILS ON GPU

Register caching

38 REGS IN KERNEL WITHOUT STENCIL

// load k and k-1 planes into registers
double xc0 = x[ijk - kStride];
double xcl = x[ijk]; ...

for(k=0; k<dimz; k++) {
    // load k+1 plane into registers
    xc2 = x[ijk + kStride]; ...

    // apply operator
    const double Ax = apply_op_ijk();

    // smoother
    xo[ijk] = xcl + ...;

    // update k and k-1 planes in registers
    xc0 = xcl; xcl = xc2; ...
}

TOTAL REG USAGE: 56 FOR FV2 AND 128 FOR FV4

7-POINT STENCIL, 18 REGS

const double Ax =
- b*h2inv*(STENCIL_TWELFTH*(
    + bir1 * (xrl - xcl)
    + bic1 * (xl1 - xcl)
    + bj1l * (xul - xcl)
    + hjcl * (zdl - xcl)
    + hkc2 * (xcl2 - xcl)
    + hkl1 * (xcl0 - xcl)
));

4TH ORDER STENCIL, 90 REGS

const double Ax =
- b*h2inv*(STENCIL_TWELFTH*(
    + bir1 * (xrl - xcl)
    + bic1 * (xl1 - xcl)
    + bj1l * (xul - xcl)
    + hjcl * (zdl - xcl)
    + hkc2 * (xcl2 - xcl)
    + hkl1 * (xcl0 - xcl)
    + 0.25*STENCIL_TWELFTH*(
        + (bid - biu) * (xld - xd1 - xlu + xu1)
        + (bic2 - bic0) * (xl2 - xc2 - x10 + xc0)
        + (bbr - bjl) * (xru - xrl - xlu + xl1)
        + (bbr - bjl) * (xru - xrl - xlu + xl1)
        + (bbr - bjl) * (xru - xrl - xlu + xl1)
        + (bbr - bjl) * (xru - xrl - xlu + xl1)
        + (bbr - bjl) * (xru - xrl - xlu + xl1)
        + (bbr - bjl) * (xru - xrl - xlu + xl1)
));

up to 1.5x speed-up!

Higher register usage may result in reduced occupancy => trade off (run experiments!)
THE EFFECT OF REGISTER CACHING

Transactions for cached loads reduced by a factor of 8

Memory utilization still high, but transferring more useful data. Still future optimization potential?

<table>
<thead>
<tr>
<th>L2 Cache</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reads</td>
<td>49131162</td>
<td>698.046</td>
<td>GB/s</td>
<td>Writes</td>
<td>4993759</td>
<td>70.95</td>
<td>GB/s</td>
<td>Total</td>
<td>54124921</td>
<td>768.997</td>
<td>GB/s</td>
</tr>
</tbody>
</table>

| Device Memory |          |          |          |          |          |          |          |          |          |          |          |          |
|               | Reads    | 29359521 | 417.135  | GB/s     | Writes   | 4380197  | 62.233  | GB/s     | Total    | 33739718 | 470.368  | GB/s     |

Kernel | Time | Speedup
---|------|--------
Original Version | 0.109443s | 1.00x
Better Memory Accesses | 0.076051s | 1.44x
Register Caching | 0.065127s | 1.68x
GPU SM ARCHITECTURE

Pascal SM

GP100
- CUDA Cores: 64
- Register File: 256 KB
- Shared Memory: 64 KB

Functional Units (CUDA cores)
- Unified Cache
- Shared Memory
- Constant Cache

SM
- 56 SMs on Tesla P100
TEX/L1

- Maxwell and Pascal: **Unified** tex/L1 cache
- Global loads are cached by default (-dlcm=ca by default)
  - 32B transaction.
  - 128B in K40. For scattered access, no need to turn L1 off to reduce transaction size.
- On GP104, default is uncached
  - To ensure caching on both GP100 and GP104, use __ldg
- Selective caching to reduce thrashing
  - Use -dlcm=cg to turn off L1 caching.
  - Add __ldg explicitly to selected variables
SHARED MEMORY

- Programmer-managed cache
- Great for caching data reused across threads in a CTA
- 64KB per SM.
  - Each block can use at most 48KB.
  - No longer split with L1. Previous call to cudaDeviceSetCacheConfig will just be ignored on Pascal

```c
__global__ void sharedMemExample(int *d, int n) {
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1; s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```
### Category: Bandwidth Bound - Register Caching

### Problem:
Data is reused within threads and memory bw utilization is high

### Goal:
Reduce amount of data traffic to/from global mem

### Indicators:
- High device memory usage, latency exposed
- Data reuse within threads and small-ish working set
- Low arithmetic intensity of the kernel

### Strategy:
- Assign registers to cache data
- Avoid storing and reloading data (possibly by assigning work to threads differently)
- Avoid register spilling
<table>
<thead>
<tr>
<th>Category:</th>
<th>Latency Bound - Texture Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Load/Store Unit becomes bottleneck</td>
</tr>
<tr>
<td>Goal:</td>
<td>Relieve Load/Store Unit from read-only data</td>
</tr>
<tr>
<td>Indicators:</td>
<td>High utilization of Load/Store Unit, pipe-busy stall reason, significant amount of read-only data</td>
</tr>
</tbody>
</table>
| Strategy:         | Load read-only data through Texture Units:  
|                   |   • Annotate read-only pointers with const __restrict__  
<p>|                   |   • Use __ldg() intrinsic |</p>
<table>
<thead>
<tr>
<th>Category:</th>
<th>Device Mem Bandwidth Bound - Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Too much data movement</td>
</tr>
<tr>
<td>Goal:</td>
<td>Reduce amount of data traffic to/from global mem</td>
</tr>
<tr>
<td>Indicators:</td>
<td>Higher than expected memory traffic to/from global memory</td>
</tr>
<tr>
<td></td>
<td>Low arithmetic intensity of the kernel</td>
</tr>
<tr>
<td>Strategy:</td>
<td>(Cooperatively) move data closer to SM:</td>
</tr>
<tr>
<td></td>
<td>• Shared Memory</td>
</tr>
<tr>
<td></td>
<td>• (or Registers)</td>
</tr>
<tr>
<td></td>
<td>• (or Constant Memory)</td>
</tr>
<tr>
<td></td>
<td>• (or Texture Cache)</td>
</tr>
</tbody>
</table>
### Category: Shared Mem Bandwidth Bound - Shared Memory

<table>
<thead>
<tr>
<th>Problem:</th>
<th>Shared memory bandwidth bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal:</td>
<td><strong>Reduce</strong> amount of data traffic to/from global mem</td>
</tr>
<tr>
<td>Indicators:</td>
<td>Shared memory loads or stores saturate</td>
</tr>
</tbody>
</table>
| Strategy: | Reduce Bank Conflicts (insert padding)  
Move data from shared memory into registers  
Change data layout in shared memory |
ITERATION 3: KERNELES WITH INCREASED ARITHMETIC INTENSITY
**HPGMG**

4th order vs 2nd order

Performs 4x the FP operations

MPI: sends 3x the messages, doubles the size (2-deep halos)

DRAM memory footprint is the same (assuming no overfetch)

Attains lower relative residual: $\sim 10^{-9}$ for a single F-cycle
FUNCTION UNIT UTILIZATION
AND STALL REASONS

Execution Dependencies
starting to become significant!

Functional units are not the bottlenecks in HPGMG, even with higher order stencils!
INSTRUCTION THROUGHPUT

Schedulers saturated

- Sched: 90%
- Shared Mem: 11%
- Texture: 8%
- Control Flow: 65%

FU saturated

- Sched: 64%
- Shared Mem: 78%
- Texture: 4%
- Control Flow: 24%

Schedulers and FU saturated

- Sched: 92%
- Shared Mem: 27%
- Texture: 6%
- Control Flow: 4%
- ALU: 90%
INSTRUCTION THROUGHPUT

Each SM has 4 schedulers (Maxwell)

Schedulers issue instructions to function units

Each scheduler schedules up to 2 instructions per cycle

A scheduler issues instructions from a single warp

Cannot issue to a pipe if its issue slot is full
STALL REASONS: EXECUTION DEPENDENCY

Memory accesses may influence execution dependencies
- Global accesses create longer dependencies than shared accesses
- Read-only/texture dependencies are counted in Texture
- Instruction level parallelism can reduce dependencies

```
a = b + c;  // ADD
a = b[i];   // LOAD
```
```
d = a + e;  // ADD
d = a + e;  // ADD
```
```
a = b + c;  // Independent ADDs
d = e + f;
```
**ILP AND MEMORY ACCESSES**

No ILP

```c
float a = 0.0f;
for( int i = 0 ; i < N ; ++i )
a += logf(b[i]);
```

2-way ILP (with loop unrolling)

```c
float a, a0 = 0.0f, a1 = 0.0f;
for( int i = 0 ; i < N ; i += 2 )
{
a0 += logf(b[i]);
a1 += logf(b[i+1]);
}
a = a0 + a1
```

#pragma unroll is useful to extract ILP
Manually rewrite code if not a simple loop
<table>
<thead>
<tr>
<th>Category:</th>
<th>Latency Bound - Instruction Level Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Not enough independent work per thread</td>
</tr>
<tr>
<td>Goal:</td>
<td>Do more parallel work inside single threads</td>
</tr>
<tr>
<td>Indicators:</td>
<td>High execution dependency, increasing occupancy has no/little positive effect, still registers available</td>
</tr>
<tr>
<td>Strategy:</td>
<td>• Unroll loops (#pragma unroll)</td>
</tr>
<tr>
<td></td>
<td>• Refactor threads to compute n output values at the same time (code duplication)</td>
</tr>
<tr>
<td>Category:</td>
<td>Compute Bound - Algorithmic Changes</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>Problem:</td>
<td>GPU is computing as fast as possible</td>
</tr>
<tr>
<td>Goal:</td>
<td>Reduce computation if possible</td>
</tr>
<tr>
<td>Indicators:</td>
<td>Clearly compute bound problem, speedup only with less computation</td>
</tr>
</tbody>
</table>
| Strategy:          | • Pre-compute or store (intermediate) results  
|                    | • Trade memory for compute time       
|                    | • Use a computationally less expensive algorithm 
|                    | • Possibly: run with low occupancy and high ILP |
SUMMARY
SUMMARY

Performance Optimization is a Constant Learning Process

1. Know your application
2. Know your hardware
3. Know your tools
4. Know your process
   • Identify the Hotspot
   • Classify the Performance Limiter
   • Look for indicators
5. Make it so!
REFERENCES

CUDA Documentation


Parallel Forall devblog

http://devblogs.nvidia.com/parallelforall/

Upcoming GTC 2017 Sessions:

S7132 - New CUDA Features and Beyond, Wed 2:30 PM
S7824 - Developer Tools Update, Wed 4:00 PM
S7495 - Optimizing Application Performance with CUDA Profiling Tools, Thur 10:00 AM
THANK YOU

JOIN THE NVIDIA DEVELOPER PROGRAM AT
developer.nvidia.com/join