S8630 - WHAT THE PROFILER IS TELLING YOU: OPTIMIZING GPU KERNELS

Jakob Progsch, Mathias Wagner
GTC 2018
BEFORE YOU START
The five steps to enlightenment

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

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

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

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 HARDWARE: VOLTA ARCHITECTURE
VOLTA V100 FEATURES

Volta Architecture
Most Productive GPU

Improved NVLink & HBM2
Efficient Bandwidth

Volta MPS
Inference Utilization

Improved SIMT Model
New Algorithms

Tensor Core
120 Programmable TFLOPS Deep Learning
# GPU COMPARISON

<table>
<thead>
<tr>
<th></th>
<th>P100 (SXM2)</th>
<th>V100 (SXM2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double/Single/Half TFlop/s</td>
<td>5.3/10.6/21.2</td>
<td>7.8/15.7/125 (TensorCores)</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/s)</td>
<td>732</td>
<td>900</td>
</tr>
<tr>
<td>Memory Size</td>
<td>16GB</td>
<td>16GB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>4096 KB</td>
<td>6144 KB</td>
</tr>
<tr>
<td>Base/Boost Clock (Mhz)</td>
<td>1328/1480</td>
<td>1312/1530</td>
</tr>
<tr>
<td>TDP (Watts)</td>
<td>300</td>
<td>300</td>
</tr>
</tbody>
</table>
VOLTA GV100 SM

GV100

- FP32 units: 64
- FP64 units: 32
- INT32 units: 64
- Tensor Cores: 8
- Register File: 256 KB
- Unified L1/Shared memory: 128 KB
- Active Threads: 2048
IMPROVED L1 CACHE

Pascal SM
- Load/Store Units
- Shared Memory 64 KB
- L1$ 24 KB
- Low Latency
- Streaming

Volta SM
- Load/Store Units
- L1$ and Shared Memory 128 KB
- L2$ 6 MB
KNOW YOUR TOOLS: PROFILERS
PROFILING TOOLS
Many Options!

From NVIDIA
• nvprof
• NVIDIA Visual Profiler (nvvp)
• Nsight Visual Studio Edition

Coming Soon:
• NVIDIA Nsight Systems
• NVIDIA Nsight Compute

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

Timeline

Summary

Guide

Analysis Results
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/
MAKE IT SO:
ITERATION 1
2ND ORDER 7-POINT STENCIL
Identify the hotspot: smooth_kernel()
IDENTIFY PERFORMANCE LIMITER

For device "Tesla V100-PCIE-16GB" the kernel’s compute utilization is significantly lower than its memory utilization. This implies 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.
Memory Utilization vs Compute Utilization

Four possible combinations:

1. **Compute Bound**
   - Comp: High
   - Mem: Low

2. **Bandwidth Bound**
   - Comp: High
   - Mem: High

3. **Latency Bound**
   - Comp: Low
   - Mem: High

4. **Compute and Bandwidth Bound**
   - Comp: High
   - Mem: High
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.
Kernel Performance Is Bound By Memory Bandwidth

For device "Tesla V100-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.
The profiler warns about low occupancy.

Limited by block size of only 8x4 = 32 threads.

**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.
Each SM has limited resources:

- max. 64K Registers (32 bit) distributed between threads
- max. 48KB (96KB opt in) 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)
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

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 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] |

**Transactions Per Access Down**

87 Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [4718592 L2 transactions for 524288 total executions]

87 Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [4718592 L2 transactions for 524288 total executions]

**Memory Utilization Up**

**Kernel**

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Version</td>
<td>2.079ms</td>
<td>1.00x</td>
</tr>
<tr>
<td>Better Memory Accesses</td>
<td>1.756ms</td>
<td>1.18x</td>
</tr>
</tbody>
</table>

+10%
### Perf-Opt Quick Reference Card

#### Latency Bound - Occupancy

<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%)  Execution Dependency High</td>
</tr>
<tr>
<td>Strategy</td>
<td>Increase occupancy by:  - Varying block size  - Varying shared memory usage  - Varying register count (use __launch_bounds)</td>
</tr>
</tbody>
</table>

#### Latency Bound - Coalescing

<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>
<tr>
<td>Strategy</td>
<td>Improve memory coalescing by:  - Cooperative loading inside a block  - Change block layout  - Aligning data  - Changing data layout to improve locality</td>
</tr>
</tbody>
</table>

#### Bandwidth Bound - Coalescing

<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 useful data per request</td>
</tr>
<tr>
<td>Indicators</td>
<td>Low global load/store efficiency, High #transactions/#request compared to ideal</td>
</tr>
<tr>
<td>Strategy</td>
<td>Improve memory coalescing by:  - Cooperative loading inside a block  - Change block layout  - Aligning data  - Changing data layout to improve locality</td>
</tr>
</tbody>
</table>
ITERATION 2: DATA MIGRATION
### PAGE FAULTS

#### Details

<table>
<thead>
<tr>
<th>GPU Page Faults</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>26051</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>0x2aaacfc21000</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Page Faults</td>
<td>18</td>
</tr>
<tr>
<td>Process</td>
<td>26051</td>
</tr>
</tbody>
</table>

#### GPU Page Faults

<table>
<thead>
<tr>
<th>Start</th>
<th>31.23757 s (31,237,569,553 ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>End</td>
<td>31.23762 s (31,237,620,177 ns)</td>
</tr>
<tr>
<td>Duration</td>
<td>50.624 μs</td>
</tr>
<tr>
<td>Memory Access Type</td>
<td>Read</td>
</tr>
</tbody>
</table>

#### Profiling Overhead

- Unified Memory
  - *CPU Page faults*
  - *Threading Throttling*
  - *Tesla V100 PCE* 16GB
  - Unified Memory
  - *Data Migration (Disk)*
    - *GPU Page faults*
    - *Data Migration (Host)*
    - *Threading Throttling*
    - *CUDA (CLDA)*
    - *Memcpy (Deu8)*
    - *Compute*
      - *21.7% void smooth_kernel_int...
      - *19.8% void copy_block_kernel...
      - *15.0% void vector kernel_for...

---

32
MEMORY MANAGEMENT
Using Unified Memory

No changes to data structures
No explicit data movements
Single pointer for CPU and GPU data

Use \texttt{cudaMallocManaged} for allocations
Solution: allocate the first CPU level with cudaMallocHost (zero-copy memory)
PAGE FAULTS
Almost gone
# PAGE FAULTS

**Significant speedup for affected kernel**

```plaintext
void interpolation_v2_kernel<int=4, int=1>(level_type, i)

| Queued   | n/a            |
| Submitted| n/a            |
| Start    | 31.40143 s (31,401,428,910 ns) |
| End      | 31.40157 s (31,401,570,860 ns) |

| Duration | 141.95 µs |
| Stream   | Default   |
| Grid Size| [1,1,8]   |
| Block Size| [8,4,1]   |
| Registers/Thread | 56 |
| Shared Memory/Bloc 0 B | |
| Launch Type | Normal |
| Occupancy Theoretical | 50% |
| Occupancy Shared Memory Confli | |
| Shared Memory Re 0 B | |
| Shared Memory Ext 96 KiB | |
| Shared Memory B1 4 B | |
```

```plaintext
void interpolation_v2_kernel<int=4, int=1>(level_type, i)

| Queued   | n/a          |
| Submitted| n/a          |
| Start    | 31.16898 s (31,168,980,770 ns) |
| End      | 31.16901 s (31,169,011,713 ns) |

| Duration | 30.943 µs |
| Stream   | Default   |
| Grid Size| [1,1,8]   |
| Block Size| [8,4,1]   |
| Registers/Thread | 56 |
| Shared Memory/Bloc 0 B | |
| Launch Type | Normal |
| Occupancy Theoretical | 50% |
| Occupancy Shared Memory Confli | |
| Shared Memory Re 0 B | |
| Shared Memory Ext 96 KiB | |
| Shared Memory B1 4 B | |
```
MEM ADVICE API

Not used here

cudaMemPrefetchAsync(ptr, length, destDevice, stream)

  Migrate data to destDevice: overlap with compute
  Update page table: much lower overhead than page fault in kernel
  Async operation that follows CUDA stream semantics

cudaMemAdvise(ptr, length, advice, device)

  Specifies allocation and usage policy for memory region
  User can set and unset at any time
ITERATION 3:
REGISTER OPTIMIZATION AND CACHING
LIMITER: STILL MEMORY BANDWIDTH

**Kernel Performance Is Bound By Memory Bandwidth**

For device "Tesla V100-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.

**Device Memory**

<table>
<thead>
<tr>
<th></th>
<th>Reads</th>
<th>Writes</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>36453066</td>
<td>4573069</td>
<td>41026135</td>
</tr>
<tr>
<td></td>
<td>664.593 GB/s</td>
<td>83.374 GB/s</td>
<td>747.966 GB/s</td>
</tr>
</tbody>
</table>
GPU MEMORY HIERARCHY

V100

- Registers (256 KB/SM): good for intra-thread data reuse
- Shared mem / L1$ (128 KB/SM): good for explicit intra-block data reuse
- L2$ (6144 KB): implicit data reuse
CACHING IN REGISTERS
No data loaded initially
CACHING IN REGISTERS
Load first set of data
CACHING IN REGISTERS

Perform calculation

Stencil
CACHING IN REGISTERS

Naively load next set of data?
CACHING IN REGISTERS
Reusing already loaded data is better
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 less redundant data

<table>
<thead>
<tr>
<th>Device Memory</th>
<th>Reads</th>
<th>29570665</th>
<th>636.46 GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes</td>
<td>4540978</td>
<td>97.737 GB/s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>34111643</td>
<td>734.197 GB/s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Version</td>
<td>2.079ms</td>
<td>1.00x</td>
</tr>
<tr>
<td>Better Memory Accesses</td>
<td>1.756ms</td>
<td>1.18x</td>
</tr>
<tr>
<td>Register Caching</td>
<td>1.486ms</td>
<td>1.40x</td>
</tr>
</tbody>
</table>
**SHARED MEMORY**

- Programmer-managed cache
- Great for caching data reused across threads in a CTA
- 128KB split between shared memory and L1 cache per SM
  - Each block can use at most 96KB shared memory on GV100
  - Search for `cudaFuncAttributePreferredSharedMemoryCarveout` in the docs

```c
__global__ void sharedMemExample(int *d) {
    __shared__ float s[64];
    int t = threadIdx.x;
    s[t] = d[t];
    __syncthreads();
    if(t>0 && t<63)
        stencil[t] = -2.0f*s[t] + s[t-1] + s[t+1];
}
```
<table>
<thead>
<tr>
<th>Category:</th>
<th>Bandwidth Bound - Register Caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Data is reused within threads and memory bw utilization is high</td>
</tr>
<tr>
<td>Goal:</td>
<td>Reduce amount of data traffic to/from global mem</td>
</tr>
<tr>
<td>Indicators:</td>
<td>High device memory usage, latency exposed Data reuse within threads and small-ish working set Low arithmetic intensity of the kernel</td>
</tr>
</tbody>
</table>
| 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__  
             • Use __ldg() intrinsic |

<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 Low arithmetic intensity of the kernel</td>
</tr>
</tbody>
</table>
| Strategy: | (Cooperatively) move data closer to SM:  
             • Shared Memory  
             • (or Registers)  
             • (or Constant Memory)  
             • (or Texture Cache) |

<table>
<thead>
<tr>
<th>Category:</th>
<th>Shared Mem Bandwidth Bound - Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>Shared memory bandwidth bottleneck</td>
</tr>
<tr>
<td>Goal:</td>
<td>Reduce 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 4:
KERNELS WITH INCREASED ARITHMETIC INTENSITY
OPERATIONAL INTENSITY

- Operational intensity = arithmetic operations/bytes written and read
- Our stencil kernels have very low operational intensity
- It might be beneficial to use a different algorithm with higher operational intensity.
- In this case this might be achieved by using higher order stencils
ILP VS OCCUPANCY

• Earlier we looked at how occupancy helps hide latency by providing independent threads of execution.

• When our code requires many registers the occupancy will be limited but we can still get instruction level parallelism inside the threads.

• Occupancy is helpful to achieving performance but not always required

• Some algorithms such as matrix multiplications allow increases in operational intensity by using more registers for local storage while simultaneously offering decent ILP. In these cases it might be beneficial to maximize ILP and operational intensity at the cost of occupancy.

\[
\begin{align*}
a &= b + c; \\
d &= a + f;
\end{align*}
\]

\[
\begin{align*}
a &= b + c; \\
d &= e + f;
\end{align*}
\]
### PERF-OPT QUICK REFERENCE CARD

<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>
</tbody>
</table>
| Strategy:              | • Unroll loops (#pragma unroll)  
                         | • Refactor threads to compute n output values at the same time (code duplication) |

### PERF-OPT QUICK REFERENCE CARD

<table>
<thead>
<tr>
<th>Category:</th>
<th>Compute Bound - Algorithmic Changes</th>
</tr>
</thead>
<tbody>
<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


NVIDIA Developer Blog

http://devblogs.nvidia.com/

Pointers to GTC 2018 Sessions:

S8718 - Optimizing HPC Simulation and Visualization Codes using the NVIDIA System Profiler (previous talk, check out recording)
S8430 - Everything You Need to Know About Unified Memory (Tue, 4:30PM)
S8106 - Volta: Architecture and Performance Optimization (Thur, 10:30 AM)
S8481 - CUDA Kernel Profiling: Deep-Dive Into NVIDIA's Next-Gen Tools (Thur, 11:00 AM)
THANK YOU

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