Profiling & Tuning Applications

CUDA Course

István Reguly
Introduction

• Why is my application running slow?
• Work it out on paper
• Instrument code
• Profile it
  • NVIDIA Visual Profiler
    • Works with CUDA, needs some tweaks to work with OpenCL
  • nvprof – command line tool, can be used with MPI applications
Identifying Performance Limiters

• CPU: Setup, data movement
• GPU: Bandwidth, compute or latency limited
• Number of instructions for every byte moved
  • ~6.4 : 1 on Kepler
  • ~7.7 : 1 on Volta
• Algorithmic analysis gives a good estimate
• Actual code is likely different
  • Instructions for loop control, pointer math, etc.
  • Memory access patterns
• How to find out?
  • Use the profiler (quick, but approximate)
  • Use source code modification (takes more work)
Analysis with Source Code Modification

• Time memory-only and math-only versions
  • Not so easy for kernels with data-dependent control flow
  • Good to estimate time spent on accessing memory or executing instructions

• Shows whether kernel is memory or compute bound

• Put an “if” statement depending on kernel argument around math/mem instructions
  • Use dynamic shared memory to get the same occupancy
__global__ void kernel(float *a) {
    int idx = threadIdx.x + blockDim.x+blockIdx.x;
    float my_a;
    my_a = a[idx];
    for (int i = 0; i < 100; i++) my_a = sinf(my_a+i*3.14f);
    a[idx] = my_a;
}

__global__ void kernel(float *a, int prof) {
    int idx = threadIdx.x + blockDim.x+blockIdx.x;
    float my_a;
    if (prof & 1) my_a = a[idx];
    if (prof & 2) {
        for (int i = 0; i < 100; i++) my_a = sinf(my_a+i*3.14f);
    }
    if (prof & 1) a[idx] = my_a;
}
Example scenarios

- **Memory-bound**: Good overlap between memory and math. Latency is not a problem.

- **Math-bound**: Good overlap between memory and math.

- **Well balanced**: Good overlap between memory and math.

- **Mem and latency bound**: Poor overlap, latency is a problem.
NVIDIA Visual Profiler

• Launch with “nvvp”
• Collects metrics and events during execution
  • Calls to the CUDA API
  • Overall application:
    • Memory transfers
    • Kernel launches
  • Kernels
    • Occupancy
    • Computation efficiency
    • Memory bandwidth efficiency
• Requires deterministic execution!
Meet the test setup

• 2D gaussian blur with a 5x5 stencil\[ \frac{1}{273} \]
• 4096^2 grid

```c
__global__ void stencil_v0(float *input, float *output,
                          int sizex, int sizey) {

    const int x = blockIdx.x*blockDim.x + threadIdx.x + 2;
    const int y = blockIdx.y*blockDim.y + threadIdx.y + 2;
    if ((x >= sizex-2) || (y >= sizey-2)) return;
    float accum = 0.0f;
    for (int i = -2; i < 2; i++) {
        for (int j = -2; j < 2; j++) {
            accum += filter[i+2][j+2]*input[sizey*(y+j) + (x+i)];
        }
    }
    output[sizey*y+x] = accum/273.0f;
}
```
Meet the test setup

• NVIDIA K80
  • GK210
  • SM 3.7
  • ECC on
  • Graphics clocks at 875MHz, Memory clocks at 2505MHz

• CUDA 10
  nvcc profiling_lecture.cu -O2 -arch=sm_35 -I. -lineinfo -DIT=0
Interactive demo of tuning process
Launch a profiling session

Executable Properties
Set executable properties

<table>
<thead>
<tr>
<th>Connection:</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toolkit/Script:</td>
<td>CUDA Toolkit 10.0 (/system/software/centos_7/gpu/cuda/10.0.130/bin/)</td>
</tr>
<tr>
<td>File:</td>
<td>/home/ireguly/a.out</td>
</tr>
<tr>
<td>Working directory:</td>
<td>Enter working directory [optional]</td>
</tr>
<tr>
<td>Arguments:</td>
<td>Enter command-line arguments</td>
</tr>
<tr>
<td>Environment:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name</td>
</tr>
</tbody>
</table>

Profile child processes
First look

**Timeline**

**Summary**

**Guide**

**Analysis results**

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application, it is important to fully utilize the compute and data movement capabilities of the GPU. To do this, you should look at your application’s overall GPU usage as well as the performance of individual kernels.

- **Examine GPU Usage**
  - Determine your application’s overall GPU usage. The analysis requires an application timeline, so your application will be run once to collect that data if it is not already available.

- **Examine Individual Kernels**
  - Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.
The Timeline

Host side

API calls

Memcpy

Compute
Analysis

1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application’s overall GPU usage as well as the performance of individual kernels.

- Examine GPU Usage
  Determine your application’s overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

- Examine Individual Kernels
  Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.

- Delete Existing Analysis Information
  If the application has changed since the last analysis then the existing analysis information may be stale and should be deleted before continuing.

- Switch to unguided analysis

---

**Guided**

**Unguided**
Examine Individual Kernels

Lists all kernels sorted by total execution time: the higher the rank the higher the impact of optimisation on overall performance

<table>
<thead>
<tr>
<th>Rank</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>stencil_v0(float*, float*, int, int)</td>
</tr>
</tbody>
</table>

Initial unoptimised (v0) 12.729ms
Utilisation

Both below 60% -> Latency!

Most of it is memory ops

Let’s investigate
Latency analysis

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Band...or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "stencil_v0" is most likely limited by instruction and memory latency.

Perform Latency Analysis

The most likely bottleneck to performance for this kernel is instruction and memory latency so you should first perform instruction and memory latency analysis to determine how it is limiting performance.

Perform Compute Analysis

Perform Memory Bandwidth Analysis

Memory throttle -> perform BW analysis
### Memory Bandwidth Analysis

#### Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory.

<table>
<thead>
<tr>
<th></th>
<th>Transactions</th>
<th>Bandwidth</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1/Shared Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Loads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Local Stores</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Shared Loads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Shared Stores</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Global Loads</td>
<td>40894464</td>
<td>248.782 GB/s</td>
<td></td>
</tr>
<tr>
<td>Global Stores</td>
<td>2621440</td>
<td>16.585 GB/s</td>
<td></td>
</tr>
<tr>
<td>Atomic</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td><strong>L1/Shared Total</strong></td>
<td>43515904</td>
<td>283.367 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Reads</td>
<td>62914560</td>
<td>248.782 GB/s</td>
<td></td>
</tr>
<tr>
<td>L1 Writes</td>
<td>4194304</td>
<td>16.585 GB/s</td>
<td></td>
</tr>
<tr>
<td>Texture Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Atomic</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td>Noncoherent Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>67108864</td>
<td>283.367 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>Texture Cache</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reads</td>
<td>0</td>
<td>0 B/s</td>
<td></td>
</tr>
<tr>
<td><strong>Device Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reads</td>
<td>3756909</td>
<td>14.856 GB/s</td>
<td></td>
</tr>
<tr>
<td>Writes</td>
<td>2904475</td>
<td>11.485 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>6661384</td>
<td>25.341 GB/s</td>
<td></td>
</tr>
<tr>
<td><strong>ECC Overhead</strong></td>
<td>2451525</td>
<td>9.694 GB/s</td>
<td></td>
</tr>
</tbody>
</table>

L1 cache not used...
6-8 transactions per access – something is wrong with how we access memory

Global memory load efficiency 53.3%
L2 hit rate 96.7%

Investigate further...

Unguided
Iteration 1 – turn on L1

Quick & easy step:
Turn on L1 cache by using
-Xptxas -dlcm=ca

Memory unit is utilized, but Global Load efficiency became even worse: 20.5%

<table>
<thead>
<tr>
<th>Line / File</th>
<th>profiling_lecture.cu · /home/mgiles/reguly/cuda_course</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [ 10485760 L2 transactions for 524288 total exec ]</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4 [ 9437184 L2 transactions for 524288 total exec ]</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [ 10485760 L2 transactions for 524288 total exec ]</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4 [ 9437184 L2 transactions for 524288 total exec ]</td>
</tr>
</tbody>
</table>

Initial unoptimised (v0) 12.729ms
Enable L1 9.886ms
Cache line utilization

32 bytes (8 floats)
Unit of transaction

L1 cache disabled:
- > 32B transactions
Min 4, Max 8 transactions
Cache line utilization

128 bytes (32 floats)  
Unit of transaction

Each time a transaction requires more than 1 128B cache line: re-issue

L1 cache enabled:  
-> 128B transactions  
-> 4*32B to L2

Min 16, Max 32 transactions
Cache line utilization

L1 cache enabled:
- > 128B transactions
- > 4*32B to L2
Min 4, Max 8 transactions

128 bytes (32 floats)
Unit of transaction

2
32

.....
Iteration 2 – 32x2 blocks

Memory utilization decreased 10%
Performance almost doubles
Global Load Efficiency 50.8%

<table>
<thead>
<tr>
<th>Line / File</th>
<th>profiling_lecture.cu - /home/mgiles/ireguly/cuda_course</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 7.5, Ideal Transactions/Access = 4</td>
</tr>
<tr>
<td>25</td>
<td>Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Initial unoptimised (v0)</th>
<th>Enable L1</th>
<th>Blocksize</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (ms)</strong></td>
<td>12.729ms</td>
<td>9.886ms</td>
<td>5.07ms</td>
</tr>
</tbody>
</table>
Key takeaway

• **Latency/Bandwidth bound**
  • Inefficient use of memory system and bandwidth

• **Symptoms:**
  • Lots of transactions per request (low load efficiency)

• **Goal:**
  • Use the whole cache line
  • Improve memory access patterns (coalescing)

• **What to do:**
  • Align data, change block size, change data layout
  • Use shared memory/shuffles to load efficiently
Latency analysis

### Stall Reasons

- execution dependency
- instruction fetch
- not selected
- memory throttle
- memory dependency
- texture
- synchronization
- other
- pipe busy
- constant

### Occupancy

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Achieved</td>
<td>41.7%</td>
</tr>
<tr>
<td>Theoretical</td>
<td>50%</td>
</tr>
<tr>
<td>Limiter</td>
<td>Block Size</td>
</tr>
</tbody>
</table>
## Latency analysis

*Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM.*

<table>
<thead>
<tr>
<th>Variable</th>
<th>Achieved</th>
<th>Theoretical</th>
<th>Device Limit</th>
<th>Grid Size: [128,2048,1] (262144 blocks) Block Size: [3:]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Occupancy Per SM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Blocks</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Warps</td>
<td>26.67</td>
<td>32</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Active Threads</td>
<td>1024</td>
<td>2048</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occupancy</td>
<td>41.7%</td>
<td>50%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td><strong>Warps</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threads/Block</td>
<td>64</td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warps/Block</td>
<td>2</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block Limit</td>
<td>32</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Latency analysis

Increase the block size so more warps can be active at the same time.

Kepler:
Max 16 blocks per SM
Max 2048 threads per SM
Occupyancy – using all “slots”

- Issue instruction
- Latency of instruction
- Next instruction

Warp 1
Warp 2
Warp 3
Warp 4
Scheduler

Increase block size to 32x4

*Illustrative only, reality is a bit more complex...*
Iteration 3 – 32x4 blocks

Initial unoptimised (v0) | 12.729ms
--- | ---
Enable L1 | 9.886ms
Blocksize | 5.07ms
Blocksize 2 | 3.67ms

占用率

<table>
<thead>
<tr>
<th>占用率</th>
<th>理论值</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>计算</td>
<td>紫色</td>
<td></td>
</tr>
<tr>
<td>内存 (加载/存储指令单元)</td>
<td>蓝色</td>
<td></td>
</tr>
</tbody>
</table>

Up 10%

Full occupancy
Key takeaway

• **Latency bound – low occupancy**
• Unused cycles, exposed latency
• **Symptoms:**
  • High execution/memory dependency, low occupancy
• **Goal:**
  • Better utilise cycles by: having more warps
• **What to do:**
  • Determine occupancy limiter (registers, block size, shared memory) and vary it
Improving memory bandwidth

• L1 is fast, but a bit wasteful (128B loads)
  • 8 transactions on average (minimum would be 4)
• Load/Store pipe stressed
  • Any way to reduce the load?
• Texture cache
  • Dedicated pipeline
  • 32 byte loads
  • const __restrict__ *
  • __ldg()
Iteration 4 – texture cache

Texture Cache

<table>
<thead>
<tr>
<th>Reads</th>
<th>67108864</th>
<th>902.37 GB/s</th>
</tr>
</thead>
</table>

| Initial unoptimised (v0) | 12.729ms |
| Blocksize 2               | 3.67ms   |
| Texture cache             | 2.38ms   |
Key takeaway

- Bandwidth bound – Load/Store Unit
- LSU overutilised
- Symptoms:
  - LSU pipe utilisation high, others low
- Goal:
  - Better spread the load between other pipes: use TEX
- What to do:
  - Read read-only data through the texture cache
  - `const __restrict__` or `__ldg()`
Compute utilization could be higher (~78%)
Lots of Integer & memory instructions, fewer FP
Integer ops have lower throughput than FP
Try to amortize the cost: increase compute per byte
• Remember, GPU is in-order:
  
  a = b + c  a = b + c  
  d = a + e  d = e + f  

• Second instruction cannot be issued before first
  • But it can be issued before the first finishes – if there is no dependency

• Applies to memory instructions too – latency much higher (counts towards stall reasons)
Instruction Level Parallelism

for (j=0; j<2; j++)
    acc += filter[j] * input[x+j];

tmp = input[x+0]
acc  += filter[0] * tmp
tmp = input[x+1]
acc  += filter[1] * tmp

#pragma unroll can help ILP
Create two accumulators
Or...

for (j=0; j<2; j++) {
    acc0 += filter[j] * input[x+j];
    acc1 += filter[j] * input[x+j+1];
}

tmp = input[x+0]
acc0 += filter[0] * tmp
acc1 += filter[0] * tmp
tmp = input[x+1]
acc0 += filter[1] * tmp
acc1 += filter[1] * tmp

Process 2 points per thread
Bonus data re-use (register caching)
Iteration 5 – 2 points per thread

<table>
<thead>
<tr>
<th>Initial unoptimised (v0)</th>
<th>12.729ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture cache</td>
<td>2.38ms</td>
</tr>
<tr>
<td>2 points</td>
<td>1.61ms</td>
</tr>
</tbody>
</table>
Key takeaway

• **Latency bound – low instruction level parallelism**
• Unused cycles, exposed latency

• Symptoms:
  • High execution dependency, one “pipe” saturated

• Goal:
  • Better utilise cycles by: increasing parallel work per thread

• What to do:
  • Increase ILP by having more independent work, e.g. more than 1 output value per thread
  • #pragma unroll
Iteration 6 – 4 points per thread

<table>
<thead>
<tr>
<th>Execution Count (% of total)</th>
<th>FP32</th>
<th>FP64</th>
<th>Integer</th>
<th>Control-Flow</th>
<th>Load/store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Count</td>
<td>40%</td>
<td>20%</td>
<td>30%</td>
<td>5%</td>
<td>10%</td>
</tr>
</tbody>
</table>

168 GB/s device BW

<table>
<thead>
<tr>
<th>Initial unoptimised (v0)</th>
<th>12.729ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 points</td>
<td>1.61ms</td>
</tr>
<tr>
<td>4 points</td>
<td>1.48ms</td>
</tr>
</tbody>
</table>
Conclusions

• Iterative approach to improving a code’s performance
  • Identify hotspot
  • Find performance limiter, understand why it’s an issue
  • Improve your code
  • Repeat

• Managed to achieve a 8.6x speedup

• Shown how NVVP guides us and helps understand what the code does

• Nsight Systems and Nsight Compute – next gen profiling tools
