Profiling & Tuning Applications

István Reguly
Oxford e-Research Centre
istvan.reguly@oerc.ox.ac.uk
Overview

- Performance limiters
  - Bandwidth, computations, latency
- Using the Visual Profiler
- “Checklist”
- Case Study: molecular dynamics code
- Command-line profiling (MPI)
- Auto-tuning
Introduction

• Why is my application running slow?
• Follow on Emerald
  – module load cuda/5.0.35
• 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
  – ~4.5 : 1 with ECC on
  – ~3.6 : 1 with ECC off
• 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)
• 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
Example scenarios

- **Memory-bound**
  - Good overlap between mem-math.
  - Latency is not a problem

- **Math-bound**
  - Good overlap between mem-math.

- **Well balanced**
  - Good overlap between mem-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
  – Memory transfers
  – Kernels
    • Occupancy
    • Computations
    • Memory accesses
• Requires deterministic execution!
Visual Profiler Demo
### Concurrent kernels

<table>
<thead>
<tr>
<th>Context 1 (CUDA)</th>
<th>0.26 s</th>
<th>0.262 s</th>
<th>0.265 s</th>
<th>0.268 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemCPy (DtoH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31.2% [3] clock_...</td>
<td>clock_block3(long*, long)</td>
<td>clock_block1(long*, lon...</td>
<td>clock_block2(long*, lon...</td>
<td>clock_block0(long*, lon...</td>
</tr>
<tr>
<td>27.1% [3] clock_...</td>
<td>clock_block0(long*, lon...</td>
<td>clock_block3(long*, lon...</td>
<td>clock_block2(long*, lon...</td>
<td>clock_block1(long*, lon...</td>
</tr>
<tr>
<td>20.8% [3] clock_...</td>
<td>clock_block1(long*, lon...</td>
<td>clock_block1(long*, lon...</td>
<td>clock_block2(long*, lon...</td>
<td>clock_block0(long*, lon...</td>
</tr>
<tr>
<td>20.8% [3] clock_...</td>
<td>clock_block1(long*, lon...</td>
<td>clock_block2(long*, lon...</td>
<td>clock_block0(long*, lon...</td>
<td>clock_block2(long*, lon...</td>
</tr>
<tr>
<td>0.0% [1] sum(lo...</td>
<td>clock_block0(long*, lon...</td>
<td>clock_block0(long*, lon...</td>
<td>clock_block0(long*, lon...</td>
<td>clock_block0(long*, lon...</td>
</tr>
</tbody>
</table>

### Streams

- **Stream 7**
  - clock_block0(long*, lon... | clock_block0(long*, lon... |
- **Stream 8**
  - clock_block1(long*, lon... | clock_block2(long*, lon... | clock_block3(long*, lon... |
- **Stream 9**
  - clock_block1(long*, lon... | clock_block0(long*, lon... | clock_block3(long*, lon... | clock_block2(long*, lon... |
- **Stream 10**
  - clock_block3(long*, lon... | clock_block1(long*, lon... | clock_block2(long*, lon... | clock_block0(long*, lon... |
<table>
<thead>
<tr>
<th>Device: Tesla K20c</th>
</tr>
</thead>
</table>

### Memory
- Requested Global Load Throughput
- Requested Global Store Throughput
- DRAM Read Throughput
- DRAM Write Throughput
- Global Store Throughput
- Global Load Throughput
- Shared Memory Efficiency
- Global Memory Load Efficiency
- Global Memory Store Efficiency
- Local Memory Overhead
- Requested Read-Only Data Cache Throughput

### Instruction
- Warrens launched
- Threads launched
- Instructions executed
- Instructions issued 1
- Instructions issued 2
- Thread inst executed
- Active cycles
- Active warps
- Sm cta launched
- Not_predicated_off_thread_inst_executed

### Memory
- Fb subp0 read sectors
- Fb subp1 read sectors
- Fb subp0 write sectors
- Fb subp1 write sectors
- Shared load
Source correlation

CUDA 5.0 debug: -G, optimized: -lineinfo
How to “use” the profiler

• Understand the timeline
  – Where and when is your code
  – Add annotations to your application
  – NVIDIA Tools Extension (markers, names, etc.)

• Find “obvious” bottlenecks
• Focus profiling on region of interest
• Dive into it
Checklist

• cudaDeviceSynchronize()
  – Most API calls (e.g. kernel launch) are asynchronous
  – Overhead when launching kernels
  – Get rid of cudaDeviceSynchronize() to hide this latency
  – Timing: events or callbacks in CUDA 5.0

• Cache config 16/48 or 48/16 kB L1/shared (default is 48k shared!)
  – cudaSetDeviceCacheConfig
  – cudaFuncSetCacheConfig
  – Check if shared memory usage is a limiting factor
Checklist

• Occupancy
  – Max 1536 threads or 8 blocks per SM on Fermi (2048/16 for Kepler)
  – Limited amount of registers and shared memory
    • Max 63 registers/thread, rest is spilled to global memory (255 for K20 Keplers)
    • You can explicitly limit it (-maxregcount=xx)
    • 48kB/16kB shared/L1: don’t forget to set it
  – Visual Profiler tells you what is the limiting factor
  – In some cases though, it is faster if you don’t maximise it (see Volkov paper) -> Autotuning!
Verbose compile

• Add –Xptxas=-v

ptxas info  : Compiling entry function '_Z10fem_kernelPiS_' for 'sm_20'
ptxas info  : Function properties for _Z10fem_kernelPiS_
   856 bytes stack frame, 980 bytes spill stores, 1040 bytes spill loads
ptxas info  : Used 63 registers, 96 bytes cmem[0]

• Feed into Occupancy Calculator
Checklist

• Precision mix (e.g. 1.0 vs 1.0f) – cuobjdump
  – F2F.F64.F32 (6* the cost of a multiply)
  – IEEE standard: always convert to higher precision
  – Integer multiplications are now expensive (6*)

• cudaMemcpy
  – Introduces explicit synchronisation, high latency
  – Is it necessary?
    • May be cheaper to launch a kernel which immediately exits
  – Could it be asynchronous? (Pin the memory!)
Asynchronous Memcopy

Memcopy H2D
Memcopy D2H

Start generating data
Start copy as soon as some data is ready

Start feeding the GPU with data
Start computing on it as soon as there is enough
Case Study

• Molecular Dynamics
• ~10000 atoms
• Short-range interaction
  – Verlet lists
• Very long simulation time
  – Production code runs for ~1 month
• Gaps between kernels – get rid of cudaDeviceSynchronize() – “free” 8% speedup
- Gaps between kernels – get rid of cudaMemcpy() – “free” 8% speedup
- None of the kernels use shared memory – set L1 to 48k – “free” 10% speedup
- Gaps between kernels – get rid of `cudaDeviceSynchronize()` – “free” 8% speedup
- None of the kernels use shared memory – set L1 to 48k – “free” 10% speedup
- `forces_second_step` is 89% of runtime
Analysing kernels

- Fairly low runtime
  - Launch latency
- Few, small blocks
  - Tail
- Low theoretical occupancy
  - 63 registers/thread
  - Spills??
- L1 configuration
  Analyze all
Memory

- Low efficiency
- Overhead because of spills
- But a very low total utilization (5.66 GB/s)
- Not really a problem
Instruction

- Very high branch divergence
  - Threads in a warp doing different things
  - SIMD – all branches executed sequentially
  - Need to look into the code

- Rest is okay
Occupancy

- Low occupancy
- Achieved much lower than theoretical
  - Load imbalance, tail
- Limiter is block size
  - In this case doesn’t help, there are already too few blocks
- Structural problem
  - Need to look into the code
Structural problems

1 thread per atom
• 10k atoms – too few threads
• Force computation with each neighbor
  – Redundant computations
  – Different number of neighbors – divergence

“Interaction” based computation (2x speedup)
• Exploit symmetry
• Lots more threads, unit work per thread
Memory-bound kernels

• What can you do if a kernel is memory-bound?
• Access pattern
  – Profiler “Global Load/Store Efficiency”
  – Struct of Arrays vs. Array of Structs

\[
\begin{array}{cccccccc}
  x & y & z & x & y & z & x & y & z \\
  x & x & x & x & y & y & y & y & z
\end{array}
\]

• Cache: every memory transaction is 128 Bytes
• High occupancy required to get close to theoretical bandwidth
• Command-line profiling tool
• Text output (CSV)
  – CPU, GPU activity, trace
  – Event collection (no metrics)
• Headless profile collection
  – Can be used in a distributed setting
  – Visualise results using the Visual Profiler
Usage

- `nvprof [nvprof_args] <app> [app_args]`

  Time(%),Time,Calls,Avg,Min,Max,Name
  ,us,,us,us,us,
  58.02,104.2260,2,52.11300,52.09700,52.12900,"op_cuda_update()"
  18.92,33.98600,2,16.99300,16.73700,17.24900,"op_cuda_res()"
  18.38,33.02400,18,1.83400,1.31200,3.77600,"[CUDA memcpy HtoD]"
  4.68,8.41600,3,2.80500,2.49600,2.97600,"[CUDA memcpyDtoH]"

- Use `--query-events` to get a list of events you can profile
Distributed Profiling

- Use the nvprof-script from the eIS wiki
- `mpirun [mpirun args] nvprof-script -o out [nvprof args] <app> [app args]
  - Will create out.0, out.1... files for different processes
- Import into Visual Profiler
  - File/Import nvprof Profile
Auto-tuning

• Several parameters that affect performance
  – Block size
  – Amount of work per block
  – Application specific

• Which combination performs the best?

• Auto-tuning with Flamingo
  – #define/read the sizes, recompile/rerun combinations
Auto-tuning Case Study

- Thread cooperation on sparse matrix-vector product
  - Multiple threads doing partial dot product on the row
  - Reduction in shared memory
- Auto-tune for different matrices
  - Difficult to predict caching behavior
  - Develop a heuristic for cooperation vs. average row length
Autotuning Case Study

The graph shows the run time (in seconds) for several applications as a function of the number of cooperating threads. The applications include:

- atmosmodd (2.63)
- crankseg_2 (14.89)
- shallow_water1 (2.00)
- webbase-1M (1.76)
- cant (8.01)

The x-axis represents the number of cooperating threads, ranging from 1 to 32. The y-axis represents the run time, which is plotted on a logarithmic scale.