CUDA Performance and Profiling

James Gain, Michelle Kuttel, Sebastian Wyngaard, Simon Perkins and Jason Brownbridge

{jgain | mkuttel | sperkins | jbrownbr}@cs.uct.ac.za
swyngaard@csir.co.za

3-6 May 2011

Resources

- Manuals From nVIDIA
  - Best Practices Guide
  - Programming Guide
  - Reference Guide

CHPC
Outline

- Performance Guidelines (from Best Practice Guide)
  - Maximise Parallel Execution
  - Optimise Memory Usage
  - Optimise Arithmetic Instruction Usage

- Performance Guide Assigns Strategies 3 Categories
  - High priority
  - Medium priority
  - Low priority

Parallel Algorithms

- Amdahl's Law:
  \[ S = \frac{1}{(1 - P) + \frac{P}{N}} \]
  - S: theoretical maximum speed-up
  - P: number of parallel parts
  - N: number of processors/cores
  - Don’t look for the impossible
  - If P = 50%, only 2x speed-up is possible at most.
Maximise Parallel Execution

- Up to this point we have only really mentioned sequential execution
  - Even though the GPU is a parallel architecture, it has been working sequentially with the CPU
- CUDA Streams allow us to execute host and device code concurrently
- Requires the programmer to understand concurrency
  - It is not a CUDA specific skill
  - Concepts such as synchronisation barriers

Concurrent CPU/GPU Computation

Sequential CPU/GPU Computation

Asynchronous CPU/GPU Computation Using 3 Streams
Asynchronous CUDA

- Asynchronous calls for:
  - Executing Kernels
    - A cudaStream is passed as a kernel parameter
  - Memory Operations
    - cudaMemcpyAsync
    - cudaMemcpyAsync
  - Functions that allocate memory
    - cudaMemcpyAsync

CUDA Streams Code Snippets

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```
CUDA Streams

- Streams have to be synchronised
  - One stream: cudaStreamSynchronize()
  - All streams: CudaThreadSynchronize()

- Refer to the programming guide

Optimised Memory Usage

- Don't waste transfers
  - Badly packed data wastes bandwidth

- For example, green data is what we want, white data is interleaved with it
- Reading from GRAM wastes 40% bandwidth because the data is not contiguous
Memory Usage

- Use memory appropriate to its usage

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location on/off chip</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off</td>
<td>No</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>All threads in block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off</td>
<td>No</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Constant</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Texture</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
</tbody>
</table>

Optimised Instruction Usage

- Use floating point and floating point SFU functions to increase instruction throughput
  - Trade-off between speed and accuracy
  - The programmer must decide

- Use Intrinsic functions
  - Faster
  - Less accurate than normal functions
  - function(): Software function
  - __function(): Hardware function
High Priority Optimisation

- Focus on parallelising sequential code
- Effective bandwidth as a performance metric
- Minimize data transfer between the host and the device
- Ensure global memory accesses are coalesced whenever possible
- Minimize the use of global memory. Prefer shared memory access where possible.
- Avoid different execution paths within the same warp

Focus on parallelising sequential code

- Think Amdahl’s Law example!!!
  - 50% of code parallelized => max speedup of 2
Effective Bandwidth

- Determine the bandwidth your CUDA implementation uses as a metric for your improvements
  
  Effective Bandwidth = \( \frac{B_r + B_w}{t} \)

- Combination of bytes read and written in time \( t \) using global memory

Minimise Transfers

- Minimise the memory transfers between host and device in either direction

- Shares the PCIE Bus
  - Only 8GB/s Bandwidth

- Use CUDA to do operations even if there is no speed-up, as long as there is not a slow down from transfers
Coalesced Transfers

- Perfect mapping from GRAM position and thread index
- Even if some threads don't, it's better than misalignment

Misaligned Reads

- If 16 threads read sequentially, but this data isn't on a 64 byte boundary
- Compute Capabilities handle it differently
  - \(<1.2\) will perform 2x 64 byte reads
  - \(\geq1.2\) will perform a 128 byte read if in the same 128 byte segment
- Data in 2 different segments requires 2 transactions
  - Halves effective bandwidth
- Using float, float2, float3, float4, SOA, and cudaMalloc helps
Use Shared Memory

- Use shared memory instead of global memory if possible
  - 100x less latency

Bank conflicts (More Medium Priority)
- 16 banks, Each thread accesses a different bank
- Simultaneously access results in serial r/w for however many threads are attempting to share

Avoiding bank conflicts
- Broadcast, 1 bank to all threads
- Each thread reads/writes a successive 32-bit value

Avoid Divergence

- All threads should execute the same code

Avoidance measures
- Make conditional statements rely on warp size
- Good example: if (tid < 2^N) { do stuff }
  - For N > 4 warps always execute the same code
  - For 2^N < 32, divergence occurs, but only in one warp
- Bad example: if (tid % 2^N == 0) { do stuff }
  - As N increases, it becomes worse
  - Only 32/ (2^N % 32) threads are active
- More idle threads == less effective GFLOPS
Medium Priority Optimisations

- Avoid shared memory bank conflicts
- Use shared memory to avoid redundant transfers from global memory
- Hide latency arising from register dependencies
- The number of threads per block should be a multiple of 32 threads
- Use the fast maths library whenever speed trumps precision

Avoid Redundant Transfers

- Load from global to shared memory once
- Prefetch to amortise latency of multiple fetches
- Removes the latency of multiple global reads
- Matrix multiplication example
  - No optimisation: 8.8 GB/s
  - Coalesced Shared Memory: 14.3 GB/s
  - Eliminate redundant reads: 29.7 GB/s
Hide Latency of Registers

- Problem: An operation using a value written to a register can only execute 24 cycles after the previous operation.
- 192 Threads per SM completely hides this:
  - $\frac{192}{24} = 8$ (the number of SPs)
  - 1 operation every cycle
  - Compute capability <1.2: 25% occupancy
  - ≥1.2: 18.75% occupancy
- Maximum threads per SM: 768 (<1.2), 1024 (≥1.2)
- Problem is that you may run out of registers.

Other Medium Priorities

- The number of threads in a block should be a multiple of 32:
  - Maps to the number of threads in a warp
- Use fast maths:
  - Use the intrinsic functions __sinf(), __expf() ....
  - Only use them if the benefit of speed outweighs accuracy.
Low Priority Optimisations

- Use zero-copy operations on integrated GPUs.
- Use shift operations to avoid expensive division and modulo calculations.
- Avoid automatic conversion of doubles to floats.

Zero Copy

- When a GPU shares its RAM with the host
  - Laptops with shared graphics RAM

- You can zero copy instead of cudaMemcpy
  - Does not cache
  - Threads can read directly from host RAM
  - Much slower than GRAM
  - Limited application
  - See the Best Practices Guide
Bitwise Operators

- Use bitwise operators

- Integer division and modulo are slow
  - Divide by 2 ($variable/2$)
  - $variable >> 1$
  - $i$ modulo $n$ ($i%n$)
    - $(i & (n-1))$

Double to Float Conversion

- The following code performs an unnecessary conversion during execution

- Assume we declare and initialise $float f$;

- Bad: $f = f + 3.0$;
  - Performs conversion from 3.0 to 3.0f at runtime
  - Costs extra cycles

- Good: $f = f + 3.0f$;
Performance Advice

- Read the Best Practices Guide
- Read the Programming Guide
- Both guides give comprehensive guides to optimise your code

Getting The Right Answer

- Accuracy is important
  - 32-bit numbers:
    - 23-bit mantissa = 7 decimal places
  - 64-bit numbers:
    - 52-bit mantissa = 16 decimal places
- Intrinsic Functions are not IEEE compliant
  - Speed vs Accuracy
  - ULP Error (Units Least Precision)
  - See Appendix in the CUDA Programming Guide
Intrinsic Functions

- Fast functions in GPU hardware (SFU)
- Prefix regular functions with __
  - __powf
  - __expf
  - __sinf, __cosf, etc...
- Execute faster than the regular functions
- Are less accurate than the regular functions
- Use them wisely

Loop Unrolling

- Explicitly writes out a loop N times
  - #pragma unroll N
- Reduces loop overhead
  - Test and increment aren’t free
Profiling

- Performance Metrics
  - Timers
  - Bandwidth

- Occupancy
  - CUDA Occupancy Calculator

- Profiling
  - CUDA Visual Profiler

Performance Metrics

- CPU Timers
  - cutil

- Use an appropriate resolution timers

- CPU timers record execution time on the host
  - Only work for blocking CUDA calls
  - Use GPU Timers for asynchronous calls
Performance Metrics

- **GPU Timers**
  - Time kernels using the GPU clock
  - Can measure execution times for asynchronous calls

Bandwidth

- Profilers measure total bandwidth usage

- Remember effective bandwidth usage:
  - Effective Bandwidth = (Br+Bw) / t

- It includes padding and wasted bits.
CUDA Tools

- CUDA Visual Profiler
- CUDA Occupancy Calculator

CUDA Visual Profiler (cudaprof)

- Helps measure and find potential problems
- GPU and CPU timing for all kernel invocations and memcpys
- Time stamps
- Access to hardware performance counters
Profiler Signals

- Events are tracked with hardware counters on signals in the chip:
  - timestamp
  - gld_incoherent
  - gld_coherent
  - gst_incoherent
  - gst_coherent
  - local_load
  - local_store
  - branch
  - divergent_branch
  - instructions – instruction count
  - warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
  - cta_launched – executed thread blocks

  Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent)

  Local loads/stores

  Total branches and divergent branches

Interpreting profiler counters

- Values represent events within a thread warp
- Only targets one multiprocessor
- Values will not correspond to the total number of warps launched for a particular kernel
- Launch enough thread blocks to ensure that target multiprocessor is given consistent percentage of the total work.
- Values are best used to identify relative performance differences between unoptimized and optimized code
- In other words, try to reduce the magnitudes of the gld/gst_incoherent, divergent branch, and warp serialize
CUDA Occupancy Calculator

- Provided in the SDK
- Use it to determine the factors limiting your code
- Use NVCC to output .cubin files which contain the information needed by the calculator:
  ```
  nvcc --ptxas-options=-v file.cu
  ```
- Occupancy increases above 50% don’t necessarily increase speed-up

Questions?
References

- Cuda Performance Slides, Ian Tunbridge, April 2010