VSCSE Summer School 2009

Many-core Processors for Science and Engineering Applications

Lecture 5: Performance Tuning
Objective

• Putting the CUDA performance knowledge to work
  – Plausible strategies may or may not lead to performance enhancement
  – Different constraints dominate in different application situations
  – Case studies help to establish intuition, idioms and ideas

• Algorithm patterns that can result in both better efficiency as well as better HW utilization

This lecture covers useful strategies for tuning CUDA application performance on many-core processors.
Computational Thinking Skills

• The ability to translate/formulate domain problems into computational models that can be solved efficiently by available computing resources
  – Understanding the relationship between the domain problem and the computational models
  – **Understanding the strength and limitations of the computing devices**
  – **Designing the model implementations to steer away from the limitations**
How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0

- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)

- However, DO NOT rely on any ordering between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results
Control Flow Instructions

• Main performance concern with branching is divergence
  – Threads within a single warp take different paths
  – Different execution paths are serialized in G80
    • The control paths taken by the threads in a warp are traversed one at a time until there is no more.

• A common case: avoid divergence when branch condition is a function of thread ID
  – Example with divergence:
    • If (threadIdx.x > 2) { }
    • This creates two different control paths for threads in a block
    • Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  – Example without divergence:
    • If (threadIdx.x / WARP_SIZE > 2) { }
    • Also creates two different control paths for threads in a block
    • Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
Parallel Reduction

• Given an array of values, “reduce” them to a single value in parallel
• Examples
  − sum reduction: sum of all values in the array
  − Max reduction: maximum of all values in the array
• Typically parallel implementation:
  − Recursively halve # threads, add two values per thread
  − Takes log(n) steps for n elements, requires n/2 threads
A Vector Reduction Example

• Assume an in-place reduction using shared memory
  – The original vector is in device global memory
  – The shared memory used to hold a partial sum vector
  – Each iteration brings the partial sum vector closer to the final sum
  – The final solution will be in element 0
A simple implementation

• Assume we have already loaded array into
  __shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
    __syncthreads();
    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t+stride];
}
Vector Reduction with Branch Divergence

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>0+1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
<td>8+9</td>
<td>10+11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0...3</td>
<td>4..7</td>
<td>8..11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..7</td>
<td>8..15</td>
<td>8..15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10–14, 2009
Some Observations

• In each iterations, two control flow paths will be sequentially traversed for each warp
  – Threads that perform addition and threads that do not
  – Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

• No more than half of threads will be executing at any time
  – All odd index threads are disabled right from the beginning!
  – On average, less than ¼ of the threads will be activated for all warps over time.
  – After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    • This can go on for a while, up to 4 more iterations (512/32=16= 2^4), where each iteration only has one thread activated until all warps retire
Shortcomings of the implementation

• Assume we have already loaded array into

```c
__shared__ float partialSum[]
```

```c
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
    stride < blockDim.x;  stride *= 2)
{
    __syncthreads();
    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t+stride];
}
```

BAD: Divergence due to interleaved branch decisions
A better implementation

• Assume we have already loaded array into
  \[
  \text{\_\_shared\_\_ float partialSum[]} \\
  \]
  
  unsigned int t = threadIdx.x;
  for (unsigned int stride = blockDim.x;
       stride > 1; stride >>= 1)
  {
    __syncthreads();
    if (t < stride)
      partialSum[t] += partialSum[t+stride];
  }
No Divergence until < 16 sub-sums
Memory Layout of a Matrix in C
Memory Coalescing

• When accessing global memory, peak performance utilization occurs when all threads in a Warp access continuous memory locations.

![Diagram showing coalesced and not coalesced memory access]

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10–14, 2009
Memory Layout of a Matrix in C

Access direction in Kernel code

Time Period 1
\[ T_1 \quad T_2 \quad T_3 \quad T_4 \]

Time Period 2
\[ T_1 \quad T_2 \quad T_3 \quad T_4 \]

\[ M \]

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10–14, 2009
Memory Layout of a Matrix in C

Access direction in Kernel code

Time Period 1

Time Period 2
Memory Access Pattern

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values
Tiled Multiply

- Make sure that tiles are all loaded in vertical patterns from the global memory
- Md data can then be accessed from shared memory in horizontal direction
Tiling Size Effects

• For good bandwidth utilization, accesses should be aligned and consist of 16 contiguous words
• Tile size 16X16 minimal required to achieve full coalescing
  – Both reduction of global memory accesses and more efficient execution of the accesses
Programmer View of Register File

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same Block only access registers assigned to itself

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009
Matrix Multiplication Example

• If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  – Each Block requires $10 \times 256 = 2560$ registers
  – $8192 = 3 \times 2560 + \text{change}$
  – So, three blocks can run on an SM as far as registers are concerned

• How about if each thread increases the use of registers by 1?
  – Each Block now requires $11 \times 256 = 2816$ registers
  – $8192 < 2816 \times 3$
  – Only two Blocks can run on an SM, **1/3 reduction of thread-level parallelism (TLP)**!!
More on Dynamic Partitioning

• Dynamic partitioning of SM resources gives more flexibility to compilers/programmers
  – One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    • This allows for finer grain threading than traditional CPU threading models.
  – The compiler can tradeoff between instruction-level parallelism and thread level parallelism
ILP vs. TLP Example

• Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, global loads have 200 cycles
  – 3 Blocks can run on each SM

• If a compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load
  – Only two can run on each SM
  – However, one only needs $200/(8\times4) = 7$ Warps to tolerate the memory latency
  – Two Blocks have 16 Warps. The performance can be actually higher!
Increase in per-thread performance, but fewer threads:
Lower overall performance in this case
Tiled Multiply

- Each **block** computes one square sub-matrix \( P_{d_{\text{sub}}} \) of size \( \text{TILE WIDTH} \)
- Each **thread** computes one element of \( P_{d_{\text{sub}}} \)
- Reduced loads from global memory (Md) to shared memory
- Reduced instruction overhead
  - More work done in each iteration
Prefetching

- One could double buffer the computation, getting better instruction mix within each thread
  - This is classic software pipelining in ILP compilers

```
Loop {
  Load current tile to shared memory
  syncthreads()
  Compute current tile
  syncthreads()
  Load next tile from global memory
}
```

```
Loop {
  Deposit current tile to shared memory
  syncthreads()
  Load next tile from global memory
  Compute current tile
  syncthreads()
}
```
Prefetch

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory

….
Instruction Mix Considerations

for (int k = 0; k < BLOCK_SIZE; ++k) 
  Pvalue += Ms[ty][k] * Ns[k][tx];

There are very few mul/add between branches and address calculation.

Loop unrolling can help.

Pvalue += Ms[ty][k] * Ns[k][tx] + …
          Ms[ty][k+15] * Ns[k+15][tx];
Unrolling

Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  // compute results for tile
  for (i = 0; i < 16; i++)
  {
    Ctemp += As[ty][i] * Bs[i][tx];
  }
  __syncthreads();
}
C[indexC] = Ctemp;

(b) Tiled Version

Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  // compute results for tile
  Ctemp +=
  As[ty][0] * Bs[0][tx];
  ...  Ctemp +=
  As[ty][15] * Bs[15][tx];
  __syncthreads();
}
C[indexC] = Ctemp;

(c) Unrolled Version

Removal of branch instructions and address calculations

Does this use more registers?
How Close Are We to Best Performance?

• Investigated applications with many optimizations
• Exhaustive optimization space search
  – Applied many different, controllable optimizations
  – Parameterized code by hand
• Hand-optimized code is deficient
  – Generally >15% from the best configuration
  – Trapped at local maxima
  – Often non-intuitive mix of optimizations
Matrix Multiplication Space

50% Performance Increase Over Hand-Optimized Version

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10–14, 2009
Some More Plausible Ideas

• One might be able to use texture memory for M accesses to reduce register usage

• Let us know if you get more than 120 GFLOPs (including CPU/GPU data transfers) for matrix multiplication. We know it can be done.
Major G80 Performance Detractors

- Long-latency operations
  - Avoid stalls by executing other threads
- Stalls and bubbles in the pipeline
  - Barrier synchronization
  - Branch divergence
- Shared resource saturation
  - Global memory bandwidth
  - Local memory capacity