Programming inverse memory hierarchy: case of stencils on GPUs

Vasily Volkov
UC Berkeley
May 18, 2010
Outline

• I got speedups in GEMM and FFT
  – 1.6x vs SGEMM in CUBLAS 1.1, now in CUBLAS
  – 3x vs CUFFT 1.1
• Lessons learned:
  – Offload storage from shared memory to registers
  – Compute few outputs per thread
• This talk:
  – Apply lessons to 3D stencils: more 2x speedups
  – Put in larger context
Need more faster memory

• Bandwidth to DRAM is limited
• Keep working set in faster on-chip memory
  – Such as shared memory on GPU
  – Access shared memory instead of DRAM if possible
• However, shared memory is small
  – G80/GT200: only 16 KB per multiprocessor
  – May be not enough
• Wouldn’t it be nice if we had more fast memory?
  – Hint: how 16,384 registers compare to shared memory?
Trend: inverse memory hierarchy (1/3)

- Per multiprocessor/SIMD on GPUs:

<table>
<thead>
<tr>
<th></th>
<th>8800GTX</th>
<th>GTX280</th>
<th>Fermi</th>
<th>HD4870</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>32KB</td>
<td>64KB</td>
<td>128KB</td>
<td>256KB</td>
</tr>
<tr>
<td>L1 storage</td>
<td>16KB</td>
<td>16KB</td>
<td>64KB</td>
<td>16KB</td>
</tr>
<tr>
<td>ratio</td>
<td>2x</td>
<td>4x</td>
<td>2x</td>
<td>16x</td>
</tr>
</tbody>
</table>

- L1 storage:
  - shared memory, L1 cache or both; LDS on ATI GPUs
  - You have more registers than shared memory
Trend: inverse memory hierarchy (2/3)

• New level of memory hierarchy on Fermi:

<table>
<thead>
<tr>
<th></th>
<th>Fermi (aggregate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>2MB</td>
</tr>
<tr>
<td>L1 storage</td>
<td>1MB</td>
</tr>
<tr>
<td>L2 storage</td>
<td>768KB</td>
</tr>
<tr>
<td>ratio</td>
<td>2/1/0.75</td>
</tr>
</tbody>
</table>

• Further from processors, but smaller
  • Usually otherwise
Trend: inverse memory hierarchy (3/3)

• Same trend on x86 architectures:

<table>
<thead>
<tr>
<th></th>
<th>Quad-core (total)</th>
<th>Larrabee (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD registers</td>
<td>1KB</td>
<td>8KB</td>
</tr>
<tr>
<td>L1 D$</td>
<td>128KB</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>8MB</td>
<td>256KB</td>
</tr>
<tr>
<td>ratio</td>
<td>1/128/8192</td>
<td>1/4/32</td>
</tr>
</tbody>
</table>

• Gap between memory hierarchy levels decreases
Why inverse?

• Single thread won’t see inverse hierarchy:
  – Up to 256B registers per thread on Fermi
  – Up to 64KB shared memory/L1 cache
  – Up to 768 KB L2 cache

• Inversion comes from parallelism
  – Now run 1024 threads on same multiprocessor
  – This is 256KB registers!

• Registers scale with SIMD and multithreading
  – Shared memory/L1 cache don’t have to
Private storage requires replication

• Can’t access shared memory on other multiprocessor
  – Get your own copy instead
• Can’t access registers in other threads
  – Get your own copy instead
  – Is it common?
  – Yes: pointers, counters, temporary values
Register usage in CUBLAS 1.1 SGEMM

Exact replicas (512x): counter, lda

Redundant storage: pointers/indices – 7 in total

Scratch space: 4 registers/thread
Optimized register usage

Do same work using fewer threads
  • Compute more outputs per thread
  • Take better advantage of space

Get much lower replication

(now in CUBLAS 2.0)
(same idea in CUFFT 2.3)
Hiding latency using fewer threads

• Less thread parallelism = poor latency hiding?
  – Not necessarily!

• Hiding latency requires memory parallelism
  – it can be supplied using thread parallelism
  – Or instruction-level parallelism

• Little’s law:
  \[ \text{data in transit [B]} = \text{latency [s]} \times \text{bandwidth [B/s]} \]

• **Same work with fewer threads = same memory parallelism**
Data dependencies in thread block may prevent overlapping latency with computation

Need several thread blocks per SM 2-4 usually suffice
• How to apply this to 3D stencils
7-point stencil in 3D

weighted average with 6 neighbors in 3D (common in FDM)

\[ W(x,y,z) := \alpha U(x,y,z) + \beta (U(x-1,y,z) + U(x+1,y,z) + U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1)) \]
Its arithmetic intensity is low

\[ W(x,y,z) := \alpha U(x,y,z) + \beta (U(x-1,y,z) + U(x+1,y,z) + U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1)) \]

- Stencil: 8 flops, 7 reads, 1 write
  - Arithmetic intensity of stencil = 1 flop/word
- GPU: 624 Gflop/s, 142 GB/s
  - Arithmetic intensity of GPU = 18 flop/word
  - Can do many more flops per each word fetched
  - ALUs will be underutilized
- That’s why we want to use fast memory
Partition the domain into blocks

Each thread block computes a block of size $XX \times YY \times ZZ$ in the domain $128^3$.
What’s the new arithmetic intensity?

Arithmetic intensity = \(\frac{4}{1+1/XX+1/YY+1/ZZ}\)

32x32x32 blocks yield AI = 3.7 flop/word

• Can’t get above 4 anyway
Circular queue technique

- Storing 3D blocks in fast memory is expensive
  - 128KB for 32x32x32 block, single precision

- Split into 2D slices
  - Need only 3 input slices to compute 1 output
Where to store the slices?

- Three 32x32 slice takes 12KB
- Only one thread block will fit
- Might need using smaller slices instead
- Or, offload storage into registers
Offload slices to registers

• Distribute slices across thread registers
• Thread \((x,y)\) gets \(U(x,y,z), U(x,y,z)\) and \(U(x,y,z)\)

\[W(x,y,z) := \alpha U(x,y,z) + \beta (U(x-1,y,z) + U(x+1,y,z) +
U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1))\]

– 4 out of 8 memory accesses are to local registers
– 4 other accesses are to registers in other threads
– But they all are to same slice

• Keep 1 slice in shared memory, 2 in registers
Compute few outputs per thread

32×32 tile requires 1024 threads if computing 1 output per thread
But only 128 threads by computing 8 elements per thread
27-point stencil

\[ W(x,y,z) = C_0 \ast U(x,y,z) + C_1 \ast (U(x-1,y,z) + U(x+1,y,z) + U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1)) + C_2 \ast (U(x-1,y-1,z) + U(x-1,y+1,z) + U(x+1,y-1,z) + U(x+1,y+1,z) + U(x,y-1,z-1) + U(x,y-1,z+1) + U(x,y+1,z-1) + U(x,y+1,z+1) + U(x-1,y,z-1) + U(x-1,y,z+1) + U(x+1,y,z-1) + U(x+1,y,z+1)) + C_3 \ast (U(x-1,y-1,z-1) + U(x-1,y-1,z+1) + U(x-1,y+1,z-1) + U(x-1,y+1,z+1) + U(x+1,y-1,z-1) + U(x+1,y-1,z+1) + U(x+1,y+1,z-1) + U(x+1,y+1,z+1)) \]

- 30 flops, 27 reads, 1 write — **1.1 flop/word**
- Arithmetic intensity = \(30/(1+(1+2/XX)*(1+2/YY)*(1+2/ZZ))\)
- For 32x32x32 this is **14 flop/word**
27-point stencil

\[ W(x,y,z) = C_0 * U(x,y,z) + C_1 * (U(x-1,y,z) + U(x+1,y,z) + \\
U(x,y-1,z) + U(x,y+1,z) + U(x,y,z-1) + U(x,y,z+1)) + \\
C_2 * (U(x-1,y-1,z) + U(x-1,y+1,z) + U(x+1,y-1,z) + \\
U(x+1,y+1,z) + U(x,y-1,z-1) + U(x,y-1,z+1) + \\
U(x,y+1,z-1) + U(x,y+1,z+1) + U(x-1,y,z-1) + \\
U(x-1,y,z+1) + U(x+1,y,z-1) + U(x+1,y,z+1)) + \\
C_3 * (U(x-1,y-1,z-1) + U(x-1,y-1,z+1) + \\
U(x-1,y+1,z-1) + U(x-1,y+1,z+1) + \\
U(x+1,y-1,z-1) + U(x+1,y-1,z+1) + \\
U(x+1,y+1,z-1) + U(x+1,y+1,z+1)) \]

- Only 4 out of 28 accesses are to local registers
- Trick: have 1 input plane, 3 output planes instead
• Performance results
Single precision, 7-point, GTX280

90 Gflop/s sustained; peak in add = 312 Gflop/s
Still very far from being compute bound
Single precision, 7-point, GTX280

- Result: 75% of peak streaming bandwidth
Single precision, 7-point, GTX280

- Count coalescing overhead (+20%): 90% of peak streaming bandwidth
- Can you do better: use larger or non-square tiles, jam stencils
Double precision, 7-point, GTX280

- Bandwidth-bound, close to peak
Double precision, 27-point, GTX280

- Faster than peak? Hidden compiler optimization
  - 120 flops gets compiled into 49 instructions
Double precision, 27-point, GTX280

- Compute bound, close to peak
Comparison with Sparse matrix-vector multiply

- Match vs. SpMV by Bell and Garland [2008]
- SpMV does not exploit stencil structure
  - More general but slower

<table>
<thead>
<tr>
<th></th>
<th>Single precision</th>
<th>Double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7-point</td>
<td>7-point</td>
</tr>
<tr>
<td>NVIDIA’s SpMV</td>
<td>0.42 ns/stencil</td>
<td>0.87 ns/stencil</td>
</tr>
<tr>
<td>Our stencil code</td>
<td>0.09 ns/stencil</td>
<td>0.21 ns/stencil</td>
</tr>
<tr>
<td>Speedup</td>
<td>4.7x</td>
<td>~4.2x</td>
</tr>
</tbody>
</table>
Red-Black Gauss Seidel

• Work by Jonathan Cohen
  – Similar to shown at last ParCFD

• Highly optimized, uses texture caches
  – 23 Gflop/s for $128^3$ volume
  – 61ms for multigrid

• My code is $\sim 2x$ faster
  – 52 Gflop/s for $128^3$ volume
  – 34ms for multigrid
Conclusion

• Look for early insights into inverse hierarchies
• May get another 2x speedup on GPU if:
  – Offload storage from shared memory to registers
  – Compute multiple outputs per thread