Optimizing GPU codes

Vasily Volkov
UC Berkeley
August 11, 2009
Background

January 2008: we got 205 Gflop/s in SGEMM on G80
  – CUBLAS was at ~128 Gflop/s
  – Best published results were at 90-100 Gflop/s
    • Baskaran et al. 2008, Ryoo et al. 2008
  – (Now our code is in CUBLAS)

That was not expected
  – How did we do that!?

June 2008: 160 Gflop/s in FFT on G80
  – CUFFT was at ~50 Gflop/s
  – (Since June 2009 CUFFT matches our performance)
Our code vs. CUBLAS 1.1

- Popular GPU programming guidelines recommend:
  - Minimize use of registers
  - Maximize use of shared memory
  - Use longer thread blocks
  - Maximize occupancy (number of concurrent instruction streams)
- CUBLAS 1.1 succeed in following all of them, but loses in performance:

<table>
<thead>
<tr>
<th></th>
<th>CUBLAS 1.1</th>
<th>Our code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers per thread</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>Shared memory per block</td>
<td>8.3 KB</td>
<td>1.1 KB</td>
</tr>
<tr>
<td>Thread block size</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>Occupancy (8800 GTX)</td>
<td>67%</td>
<td>33%</td>
</tr>
<tr>
<td>Performance (8800 GTX)</td>
<td>128 Gflop/s</td>
<td>205 Gflop/s</td>
</tr>
</tbody>
</table>

- Note that both codes do the same amount of work per block
  - $2048^2K$ flops per thread if multiplying $MxK$ matrix by $KxN$ matrix
Memory latency hiding

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

How to keep much data in transit?
- (Prefetch)
- Use long vectors: SIMD
- Use many threads: SMT
- Mixture: SIMT = SIMD + SMT

It is all about memory concurrency, not threads
Little’s law in numbers

<table>
<thead>
<tr>
<th></th>
<th>8800GTX</th>
<th>9800GTX</th>
<th>GTX280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sustained bandwidth</td>
<td>76 GB/s</td>
<td>58 GB/s</td>
<td>127 GB/s</td>
</tr>
<tr>
<td>Sustained latency</td>
<td>320 ns</td>
<td>300 ns</td>
<td>335 ns</td>
</tr>
<tr>
<td>Little’s law</td>
<td>24320 B</td>
<td>17400 B</td>
<td>42545 B</td>
</tr>
<tr>
<td>Max threads</td>
<td>12288</td>
<td>12288</td>
<td>30720</td>
</tr>
<tr>
<td>Min requests/thread</td>
<td>2.0 B</td>
<td>1.4 B</td>
<td>1.4 B</td>
</tr>
</tbody>
</table>

Hides latency if access is **very fine grain**
≥2 bytes in independent requests in thread

So small granularity may be unnecessary
Block/tiled algorithms

Workflow: **load block**, compute, **store block**, repeat
- all in one thread block
Consider 32x32 block of single precision numbers
This is **4 KB** data/block or per multiprocessor
Hides latency no matter how many threads are run:

<table>
<thead>
<tr>
<th></th>
<th>8800GTX</th>
<th>9800GTX</th>
<th>GTX280</th>
</tr>
</thead>
<tbody>
<tr>
<td># of multiprocessors</td>
<td>16</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>Little’s law</td>
<td>24320 B</td>
<td>17400 B</td>
<td>42545 B</td>
</tr>
<tr>
<td>Per multiprocessor</td>
<td><strong>1.5 KB</strong></td>
<td><strong>1.1 KB</strong></td>
<td><strong>1.4 KB</strong></td>
</tr>
</tbody>
</table>

**Tiled algorithms don’t require many threads**
## Register files

<table>
<thead>
<tr>
<th></th>
<th>8800GTX</th>
<th>9800GTX</th>
<th>GTX280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads/multiprocessor</td>
<td>768</td>
<td>768</td>
<td>1024</td>
</tr>
<tr>
<td>Registers/multiprocessor</td>
<td>8192</td>
<td>8192</td>
<td>16384</td>
</tr>
<tr>
<td>Registers/thread</td>
<td>10</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Registers, total</td>
<td>512 KB</td>
<td>512 KB</td>
<td>1.9 MB</td>
</tr>
</tbody>
</table>

Many threads require many registers
Up to ≈ 2 MB registers on die in total
- largest memory on die (4x shared memory)

Got many registers – how to use them well?
## Register usage in CUBLAS SGEMM

| warp 1 | warp 2 | warp 3 | warp 4 | warp 5 | warp 6 | warp 7 | warp 8 | warp 9 | warp 10 | warp 11 | warp 12 | warp 13 | warp 14 | warp 15 | warp 16 |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |
|        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |
|        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |
|        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |
| C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data |

### C's index

<table>
<thead>
<tr>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
<th>counter</th>
</tr>
</thead>
</table>

### B's pointer

| C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data | C's data |

### B's pointer in shared memory

<table>
<thead>
<tr>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
<th>lda</th>
</tr>
</thead>
<tbody>
<tr>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
</tr>
<tr>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
</tr>
<tr>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
<td>*****</td>
</tr>
</tbody>
</table>

### C's index

| counter | counter | counter | counter | counter | counter | counter | counter | counter | counter | counter | counter | counter | counter | counter | counter |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|

### Runs many threads

### Registers are wasted for auxiliary information

### Extensive thread parallelism eats registers fast
Register usage in our SGEMM

Run fewer threads
Half registers are used for working set

Fewer threads – fewer registers wasted
• We want to use registers for working set
• But they are not shared
  – Use distributed memory algorithms?
Local data layout in our SGEMM

- Blocks in A and C are row-cyclic distributed across threads
  - Little inter-thread communication required
Is # of thread blocks important?

CUBLA SGEMM again
- 1 thread block per multiprocessor
- Many local barriers inside
- Memory latency doesn’t overlap with computation

More thread blocks is good
- 2-4 often enough
Our code vs. CUBLAS 1.1

Performance in multiplying two $N \times N$ matrices on GeForce 8800 GTX:

- Multiply-and-add with an operand in shared memory (66%)
- Our implementation (60%)
- CUBLAS 1.1 (37%)

Our SGEMM is used in CUBLAS 2.0 and later; open-source
Performance Results

Our solution runs at \(~\text{50\%}\) of the system’s peak (shown on the right)

Open-source
• If we compute on CPU anyway, do that in parallel with computing on GPU
Similar algorithm in double precision
FFT Performance on NVIDIA 8800GTX

- **compute bound (173 GInstr/s)**
- **bandwidth bound (76 GB/s)**

![Graph showing performance of FFT on NVIDIA 8800GTX.](image)
FFT Performance on NVIDIA 9800GTX

---

**compute bound**: (214 GInstr/s)

**bandwidth bound**: (58 GB/s)

---

CUDA FFT
Also used in OpenCL FFT
SGEMM, A:Mx2048, B:2048xN

Many 512 pt FFTs

256³ 3D stencil (double precision)

Geforce GTX280

(3D stencil flop count doesn’t include CSE done by compiler)
Performance doesn’t get better after 10,000 threads
Running more threads than can fit at a time is not critical
Global synchronization

• Global synchronization $\approx$ launch new kernel
• Launch new kernel $\approx 3\div7$ $\mu$s in overhead
• LU factorization of $N\times N$ matrix $\approx 4N$ kernel invocations
  – This is $12\div28$ ms for $1000\times1000$ matrix
  – This is $24\div56$ Gflop/s upperbound
  – But you get $50$ Gflop/s on quad-core CPU
• May not worth implementing on GPU
Panel Factorization

Factorizing $N \times 64$ matrix in GPU memory using LAPACK’s SGETF2:

- When optimizing CPU-GPU communication, mind:
  - Not only #bytes transferred
  - But also #kernels called
Alternative global synchronization

• 3 μs is 10x memory latencies – can do better?
  – Why not do all work in one kernel?
• Threads can globally communicate via DRAM
• Can implement custom barrier
  – Requires no atomic operations
  – Requires memory consistency
    • Memory fence will do (available since CUDA 2.2)
• Trends: fast on-chip global synchronization
  – Coherent caches on Intel Larrabee
  – ATI GPUs have global shared memory (GDS)
Fast on-chip communication?

• GPU has a memory crossbar anyway
  – Can we use it for on-chip global communication?